导航

- 首页
- 社区主页
- 当前事件
- 最近更改
- 随机页面
- 使用帮助
- NOCOW地图
- 新手试练场

搜索

进入 搜索

工具箱

- 链入页面
- 链出更改
- 特殊页面
- 可打印版
- 永久链接

Kuhn-Munkres算法

Maigo的KM算法讲解（的确精彩）

> KM算法是通过给每个顶点一个标号（叫做顶标）来把求最大权匹配的问题转化为求完备匹配的问题的。设顶点Xi的顶标为A[i]，

顶点Yi的顶标为B[i]，顶点Xi与Yj之间的边权为w[i,j]。在算法执行过程中的任一时刻，对于任一条边(i,j)，A[i]+B[j]>=w[i,j]始终成立。KM算法的正确性基于以下定理：

* 若由二分图中所有满足A[i]+B[j]=w[i,j]的边(i,j)构成的子图（称做相等子图）有完备匹配，那么这个完备匹配就是二分图的最大权匹配。

这个定理是显然的。因为对于二分图的任意一个匹配，如果它包含于相等子图，那么它的边权和等于所有顶点的顶标和；如果它有的边不包含于相等子图，那么它的边权和小于所有顶点的顶标和。所以相等子图的完备匹配一定是二分图的最大权匹配。

初始时为了使A[i]+B[j]>=w[i,j]恒成立，令A[i]为所有与顶点Xi关联的边的最大权，B[j]=0。如果当前的相等子图没有完备匹配，就按下面的方法修改顶标以使扩大相等子图，直到相等子图具有完备匹配为止。

我们求当前相等子图的完备匹配失败了，是因为对于某个X顶点，我们找不到一条从它出发的交错路。这时我们获得了一棵交错树，它的叶子结点全部是X顶点。现在我们把交错树中X顶点的顶标全都减小某个值d，Y顶点的顶标全都增加同一个值d，那么我们会发现：

两端都在交错树中的边(i,j)，A[i]+B[j]的值没有变化。也就是说，它原来属于相等子图，现在仍属于相等子图。

两端都不在交错树中的边(i,j)，A[i]和B[j]都没有变化。也就是说，它原来属于（或不属于）相等子图，现在仍属于（或不属于）相等子图。

X端不在交错树中，Y端在交错树中的边(i,j)，它的A[i]+B[j]的值有所增大。它原来不属于相等子图，现在仍不属于相等子图。

X端在交错树中，Y端不在交错树中的边(i,j)，它的A[i]+B[j]的值有所减小。也就说，它原来不属于相等子图，现在可能进入了相等子图，因而使相等子图得到了扩大。

现在的问题就是求d值了。为了使A[i]+B[j]>=w[i,j]始终成立，且至少有一条边进入相等子图，d应该等于min{A[i]+B[j]-w[i,j]|Xi在交错树中，Yi不在交错树中}。

以上就是KM算法的基本思路。但是朴素的实现方法，时间复杂度为O(n4)——需要找O(n)次增广路，每次增广最多需要修改O(n)次顶标，每次修改顶标时由于要枚举边来求d值，复杂度为O(n2)。实际上KM算法的复杂度是可以做到O(n3)的。我们给每个Y顶点一个“松弛量”函数slack，每次开始找增广路时初始化为无穷大。在寻找增广路的过程中，检查边(i,j)时，如果它不在相等子图中，则让slack[j]变成原值与A[i]+B[j]-w[i,j]的较小值。这样，在修改顶标时，取所有不在交错树中的Y顶点的slack值中的最小值作为d值即可。但还要注意一点：修改顶标后，要把所有的slack值都减去d。

二分图最大权完美匹配KM算法

好吧，这弄点正经的。这次就写写大家肯定很久以前就搞出来的KM。我写这个是因为前几天整理模板的时候居然发现我的KM还是O(n^4)的，虽然实际运行效果大部分和O(n^3)差不多，但是理论的上界仍然让我不爽，就像network simplex algorithm一样。

先说一下KM的适用范围。据我分析KM实际上可以对任意带权（无论正负权）二分图求最大/最小权完美匹

配，唯一的一个，也是最重要的一个要求就是这个匹配必须是完美匹配，否则KM的正确性将无法得到保证。这个当了解了KM的正确性证明之后自然就会知道。非完美的匹配的似乎必须祭出mincost maxflow了。

然后就是KM的时间界。这里略去KM的步骤不谈。众所周知，KM弄不好就会写出O(n^4)的算法，而实际上是存在O(n^3)的实现的。那么O(n^4)究竟是慢在什么地方呢？这个就需要搞清楚O(n^4)的4究竟是怎么来的。

每个点都需要作一次增广，所以有一个n的循环。每个循环内部，每次可能无法得到一条增广路，需要新加入一个y顶点，然后重新寻找增广路。一次最少加进1个点，所以最多加入n次。每次重新找一遍增广路n^2，更新距离标号需要扫描每一条边n^2，所以迭加起来O(n)*O(n)*O(n^2)，结果自然就是O(n^4)。

第一层和第二层循环似乎没有很好的方法可以把它搞掉，所以我们只能从第三层，也就是每次的O(n^2)入手。这一层包括两个部分，一个是增广路的n^2，一个是更新标号的n^2，需要将二者同时搞掉才能降低总共的复杂度。注意更新标号之后有一个最重要的性质，那就是 原来存在的合法边仍然合法，更新只是将不合法的边变得合法。所以当我们找到一次交错树，没有增广路之后，下次再寻找的时候完全没有必要重新开始，因为原先有的边更新之后还有，所以完全可以接着上一次得到的交错树继续寻找。那么应该从什么地方继续号、开始搜索呢？很明显是那些新加进的点，也就是新进入的那些y点。这样虽然不知道每次更新标号会又扫描多少次，但是每条边最多也就被扫描一次，然后被添加进交错树一次。所以这一块，n层循环总的复杂度是O(n^2)。按照这样描述的话，用dfs似乎没有可以接着上一次找的方法，所以只能用bfs来写增广路的搜索了。

然后就是重标号。这一段实际上是由重新扫了一次边，然后对x在树中而y不在的边进行侦测，然后重新标号。想把这个搞掉似乎是有些困难，但是我们先做的是增广路搜索然后才是标号，增广路搜索的时候不也是要扫边么？要是能在bfs的时候记录一下什么信息，到时候直接取用不就好了？所以，做法就是在bfs的时候，对于每个扫到的这种边，更新一下对应的y顶点的标号，这个标号的意义就是y点连接的所有这种边当中可松弛的最小值，定义为slack[y]。然后每次更新的时候扫一下所有的y，对于所有没在交错树中的y，找到最小slack[y]，然后更新就可以了。注意由于我们要接着上一次进行bfs，所以上一次算出来的标号也要留下来。别忘了重标号之后每个y点的slack[y]也要同样更新，这样每次寻找标号并更新的复杂度就是O(n)了，n层重标号最多也就是O(n^2)，然后bfs的O(n^2)，增广的O(n)，所以对于每个点，想对匹配进行增广，复杂度就是O(n^2)，n个点每个点来一次自然就是O(n^3)了。

CODE:

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;

const int size = 160;
const int INF = 100000000; // 相对无穷大

bool map[size][size];         // 二分图的相等子图, map[i][j] = true 代表Xi与Yj有边
bool xckd[size], yckd[size]; // 标记在一次DFS中, Xi与Yi是否在交错树上
int match[size];             // 保存匹配信息, 其中i为Y中的顶点标号, match[i]为X中顶点标号

bool DFS(int, const int);

void KM_Perfect_Match(const int n, const int edge[][size]) {
   int i, j;
   int lx[size], ly[size];   // KM算法中Xi与Yi的标号
   for(i = 0; i < n; i++) {
      lx[i] = -INF;
      ly[i] = 0;
      for(j = 0; j < n; j++) {
         lx[i] = max(lx[i], edge[i][j]);
      }
   }
    bool perfect = false;
    while(!perfect) {
      // 初始化邻接矩阵
        for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
            if(lx[i]+ly[j] == edge[i][j]) map[i][j] = true;
            else map[i][j] = false;
         }
      }
```

```
        // 匹配过程
        int live = 0;
        memset(match, -1, sizeof(match));
        for(i = 0; i < n; i++) {
            memset(xckd, false, sizeof(xckd));
            memset(yckd, false, sizeof(yckd));
            if(DFS(i, n)) live++;
            else {
                xckd[i] = true;
                break;
            }
        }
        if(live == n) perfect = true;
        else {
            // 修改标号过程
            int ex = INF;
            for(i = 0; i < n; i++) {
                for(j = 0; xckd[i] && j < n; j++) {
                    if(!yckd[j]) ex = min(ex, lx[i]+ly[j]-edge[i][j]);
                }
            }
            for(i = 0; i < n; i++) {
                if(xckd[i]) lx[i] -= ex;
                if(yckd[i]) ly[i] += ex;
            }
        }
    }
}

// 此函数用来寻找是否有以Xp为起点的增广路径，返回值为是否含有增广路

bool DFS(int p, const int n)
{
    int i;
    for(i = 0; i < n; i++) {
        if(!yckd[i] && map[p][i]) {
            yckd[i] = true;
            int t = match[i];
            match[i] = p;
            if(t == -1 || DFS(t, n)) {
                return true;
            }
            match[i] = t;
            if(t != -1) xckd[t] = true;
        }
    }
    return false;
}

int main()
{
    int n, edge[size][size]; // edge[i][j]为连接Xi与Yj的边的权值
    int i;

    /*****************************************************
     *        在此处要做的工作 :
     *        读取二分图每两点间边的权并保存在edge[][]中,
     * 若X与Y数目不等，应添加配合的顶点
     *        保存二分图中X与Y的顶点数n,若上一步不等应保
     * 存添加顶点完毕后的n
     *****************************************************/

    KM_Perfect_Match(n, edge);
    int cost = 0;
    for(i = 0; i < n; i++) {
        cost += edge[match[i]][i];
    }
    // cost 为最大匹配的总和, match[]中保存匹配信息

    return 0;
}
```

另附O(N^3)的算法代码:

```
#include <cstdio>
#include <queue>
#include <algorithm>
using namespace std;

const int N = 128;
const int INF = 1 << 28;

class Graph {
private:
    bool xckd[N], yckd[N];
```

```
    int n, edge[N][N], xmate[N], ymate[N];
    int lx[N], ly[N], slack[N], prev[N];
    queue<int> Q;
    bool bfs();
    void agument(int);
public:
    bool make();
    int KMMatch();
};
bool Graph::make() {
    int house[N], child[N], h, w, cn = 0;
    char line[N];
    scanf("%d %d", &h, &w);
    if(w == 0) return false;
    scanf("\n"); n = 0;
    for(int i = 0; i < h; i++) {
        gets(line);
        for(int j = 0; line[j] != 0; j++) {
            if(line[j] == 'H') house[n++] = i * N + j;
            if(line[j] == 'm') child[cn++] = i * N + j;
        }
    }
    for(int i = 0; i < n; i++) {
        int cr = child[i] / N, cc = child[i] % N;
        for(int j = 0; j < n; j++) {
            int hr = house[j] / N, hc = house[j] % N;
            edge[i][j] = -abs(cr-hr) - abs(cc-hc);
        }
    }
    return true;
}
bool Graph::bfs() {
    while(!Q.empty()) {
        int p = Q.front(), u = p>>1; Q.pop();
        if(p&1) {
            if(ymate[u] == -1) { agument(u); return true; }
            else { xckd[ymate[u]] = true; Q.push(ymate[u]<<1); }
        } else {
            for(int i = 0; i < n; i++)
                if(yckd[i]) continue;
                else if(lx[u]+ly[i] != edge[u][i]) {
                    int ex = lx[u]+ly[i]-edge[u][i];
                    if(slack[i] > ex) { slack[i] = ex; prev[i] = u; }
                } else {
                    yckd[i] = true; prev[i] = u;
                    Q.push((i<<1)|1);
                }
        }
    }
    return false;
}
void Graph::agument(int u) {
    while(u != -1) {
        int pv = xmate[prev[u]];
        ymate[u] = prev[u]; xmate[prev[u]] = u;
        u = pv;
    }
}
int Graph::KMMatch() {
    memset(ly, 0, sizeof(ly));
    for(int i = 0; i < n; i++) {
        lx[i] = -INF;
        for(int j = 0; j < n; j++) lx[i] >?= edge[i][j];
    }
    memset(xmate, -1, sizeof(xmate)); memset(ymate, -1, sizeof(ymate));
    bool agu = true;
    for(int mn = 0; mn < n; mn++) {
        if(agu) {
            memset(xckd, false, sizeof(xckd));
            memset(yckd, false, sizeof(yckd));
            for(int i = 0; i < n; i++) slack[i] = INF;
            while(!Q.empty()) Q.pop();
            xckd[mn] = true; Q.push(mn<<1);
        }
        if(bfs()) { agu = true; continue; }
        int ex = INF; mn--; agu = false;
        for(int i = 0; i < n; i++)
            if(!yckd[i]) ex <?= slack[i];
        for(int i = 0; i < n; i++) {
            if(xckd[i]) lx[i] -= ex;
            if(yckd[i]) ly[i] += ex;
            slack[i] -= ex;
        }
        for(int i = 0; i < n; i++)
            if(!yckd[i] && slack[i] == 0) { yckd[i] = true; Q.push((i<<1)|1); }

    }
    int cost = 0;
```

```
    for(int i = 0; i < n; i++) cost += edge[i][xmate[i]];
    return cost;
}

int main()
{
    Graph g;

    while(g.make()) printf("%d\n", -g.KMMatch());

    return 0;
}
```

二分图匹配----基于匈牙利算法和KM算法2007-09-19 16:54设G=(V,{R})是一个无向图。如顶点集V可分割为两个互不相交的子集，并且图中每条边依附的两个顶点都分属两个不同的子集。则称图G为二分图。

```
v       给定一个二分图G，在G的一个子图M中，M的边集{E}中的任意两条边都不依附于同一个顶点，则称M是一个匹配。
```

v 选择这样的边数最大的子集称为图的最大匹配问题(maximal matching problem)

v 如果一个匹配中，图中的每个顶点都和图中某条边相关联，则称此匹配为完全匹配，也称作完备匹配。

最大匹配在实际中有广泛的用处，求最大匹配的一种显而易见的算法是：先找出全部匹配，然后保留匹配数最多的。但是这个算法的复杂度为边数的指数级函数。因此，需要寻求一种更加高效的算法。

匈牙利算法是求解最大匹配的有效算法，该算法用到了增广路的定义(也称增广轨或交错轨)：若P是图G中一条连通两个未匹配顶点的路径，并且属M的边和不属M的边(即已匹配和待匹配的边)在P上交替出现，则称P为相对于M的一条增广路径。

由增广路径的定义可以推出下述三个结论：

v 1. P的路径长度必定为奇数，第一条边和最后一条边都不属于M。

v 2. P经过取反操作（即非M中的边变为M中的边，原来M中的边去掉）可以得到一个更大的匹配M’。

v 3. M为G的最大匹配当且仅当不存在相对于M的增广路径。

从而可以得到求解最大匹配的匈牙利算法：

v (1)置M为空

v (2)找出一条增广路径P，通过取反操作获得更大的匹配M’代替M

v (3)重复(2)操作直到找不出增广路径为止

根据该算法，我选用dfs (深度优先搜索)实现。

程序清单如下：

```
int match[i]            //存储集合m中的节点i在集合n中的匹配节点,初值为-1。
int n,m,match[100];                         //二分图的两个集合分别含有n和m个元素。
bool visit[100],map[100][100];                  //map存储邻接矩阵。
bool dfs(int k)
{
int t;
for(int i = 0; i < m; i++)
     if(map[k][i] && !visit[i]){
      visit[i] = true;
      t = match[i];
      match[i] = k;                                 //路径取反操作。
      if(t == -1 || dfs(t))          //寻找是否为增广路径
       return true;
      match[i] = t;
}
     return false;
}
int main()
{
//..........

     int    s = 0;

     memset(match,-1,sizeof(match));
     for(i = 0; i < n; i++){       //以二分集中的较小集为n进行匹配较优
      memset(visit,0,sizeof(visit));
      if(dfs(i))
```

```
            s++;            //s为匹配数
        }
    //............
    return 0;
    }
```

二分图最优匹配：对于二分图的每条边都有一个权（非负），要求一种完备匹配方案，使得所有匹配边的权和最大，记做最优完备匹配。（特殊的，当所有边的权为1时，就是最大完备匹配问题）

解二分图最优匹配问题可用穷举的方法，但穷举的效率=n！，所以我们需要更加优秀的算法。

先说一个定理：设M是一个带权完全二分图G的一个完备匹配，给每个顶点一个可行顶标(第i个x顶点的可行标用lx[i]表示，第j个y顶点的可行标用ly[j]表示)，如果对所有的边(i,j) in G,都有lx[i]+ly[j]>=w[i,j]成立(w[i,j]表示边的权)，且对所有的边(i,j) in M,都有lx[i]+ly[j]=w[i,j]成立，则M是图G的一个最优匹配。

Kuhn－Munkras算法（即KM算法）流程：

v (1)初始化可行顶标的值

v (2)用匈牙利算法寻找完备匹配

v (3)若未找到完备匹配则修改可行顶标的值

v (4)重复(2)(3)直到找到相等子图的完备匹配为止

KM算法主要就是控制怎样修改可行顶标的策略使得最终可以达到一个完美匹配，首先任意设置可行顶标（如每个X节点的可行顶标设为它出发的所有弧的最大权，Y节点的可行顶标设为0），然后在相等子图中寻找增广路，找到增广路就沿着增广路增广。而如果没有找到增广路呢，那么就考虑所有现在在匈牙利树中的X节点（记为S集合），所有现在在匈牙利树中的Y节点（记为T集合），考察所有一段在S集合，一段在not T集合中的弧，取 delta = min {l(xi)+l(yj)-w(xi,yj) , | xi in S, yj in not T} 。明显的，当我们把所有S集合中的l(xi)减少delta之后，一定会有至少一条属于(S, not T)的边进入相等子图，进而可以继续扩展匈牙利树，为了保证原来属于(S,T)的边不退出相等子图，把所有在T集合中的点的可行顶标增加delta。随后匈牙利树继续扩展，如果新加入匈牙利树的Y节点是未盖点，那么找到增广路，否则把该节点的对应的X匹配点加入匈牙利树继续尝试增广。

复杂度分析：由于在不扩大匹配的情况下每次匈牙利树做如上调整之后至少增加一个元素，因此最多执行n次就可以找到一条增广路，最多需要找n条增广路，故最多执行n^2次修改顶标的操作，而每次修改顶标需要扫描所有弧，这样修改顶标的复杂度就是O(n^2)的，总的复杂度是O(n^4)的。

```
    对于not T的每个元素yj，定义松弛变量slack(yj) =min{l(xi)+l(yj)-w(xi,yj), | xi in S}，很明显每次的
```

delta = min{slack(yj), | yj in not T}，每次增广之后用O(n^2)的时间计算所有点的初始slack，由于生长匈牙利 树的时候每条弧的顶标增量相同，因此修改每个slack需要常数时间（注意在修改顶标后和把已盖Y节点对应的X节点加入匈牙 利树的时候是需要修改slack的）。这样修改所有slack值时间是O(n)的，每次增广后最多修改n次顶标，那么修改顶标的总 时间降为O(n^2)，n次增广的总时间复杂度降为O(n^3)。事实上这样实现之后对于大部分的数据可以比O(n^4)的算法快一倍 左右。

利用二分图匹配的匈牙利算法和KM算法，我们可以求解大部分的关于二分图的问题，它们提供了求解最大匹配和最优匹配的有 效算法，在具体编程时我们只要多注意优化，我们就可以得出求解这类问题的有效方法，从而可以对这类实际问题进行有效合 理的解决