****

算法设计结课论文

最大团问题的求解性能比较

学生：郎文翀 | 指导老师：王征 | 日期：2020.12.25

实验目的

掌握利用算法进行问题求解的实验流程

强化对课堂所学算法原理的理解

提升利用算法原理进行编码实践的能力

了解算法研究的前沿动态，拓展算法改进的思路

实验课题

2.3算法求解类问题4.5对于最大团问题利用动态规划，回溯和分枝限界三种算法进行求解并分析不同算法解决同一问题的复杂度，归纳不同求解算法的优缺点。

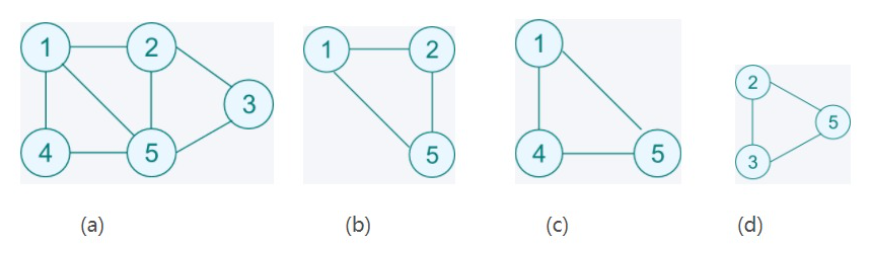
研究问题描述

给定无向图G求解G中的最大集团大小（用顶点数量表示），并给出任意一组最大团的顶点组合同时分析采取的不同方法的时间复杂度和算法性能的比较。

最大集团定义

1. 完全图：如果无向图中的任何一对顶点之间都有一条边，这种无向图称为完全图。
2. 完全子图：给定无向图G=（V,E），如果点集U⊆V，且对任意U中的两个点u,v都有（u,v)⊆E,则成U是G的完全子图。
3. 团（最大完全子图）：U是G的团当且仅当U不包含在G的更大的完全子图中。
4. 最大团：G的最大团是指G中所含顶点数最多的团。
5. 独立集：对于给定无向图G=（V,E），如果顶点集合V*⊆V，若V\*中任何两个顶点均不相连，则称V\*为G的独立集。
6. 最大独立集：G中所含顶点数最多的独立集。
7. 补图：完全图中未相连的边和无向图的所有点所构成的图就成为补图

例如：对于图



a是一个无向图，b,c,d都是a的最大团。b就是a的一个补图，因为（1,3）（2,4）没有在无向图中存在，所以他是补图的边，而且有如下关系：无向图G的团和G补图的独立集之间存在一一对应的关系，特别地，U是G的最大团当且仅当U是G补图的最大独立集。如{1,2,5}是G的最大团，同时{1,2,5}也是G补图的最大独立集。

回溯法代码设计流程

# 一、设计思路

无向图G的最大子团和最大独立子集问题实际上是一种题型，两者之间可以互相转换，并且使用回溯法可以将时间复杂度控制在O(n\*2^n)以内解决。因为最大团和最大独立子集问题都可以看做是图G顶点集V的子集选取问题，因此，都可以选择使用状态空间树展开求解，解空间就是n元组。解决思想是，设当前的E-节点（又称为扩展节点)Z处于解空间的第i层，在进入左子树，即代表1状态，加入Z之前，我们需要先进行判断他是否和已选择的所有团节点都连接，如果没有，则不能进入，在进入右子树即0状态不选取Z选取除Z以外的其他点之前判断是否还有足够多个点可以供我们选择来使得有可能在右子树（即选择Z以外剩余点）形成比已知最大团更大的团。再具体实现时，使用邻接矩阵表示图G，以一个整数型数组向量返回找到最大团的解，v[i]=1表示顶点i属于最大团的顶点。

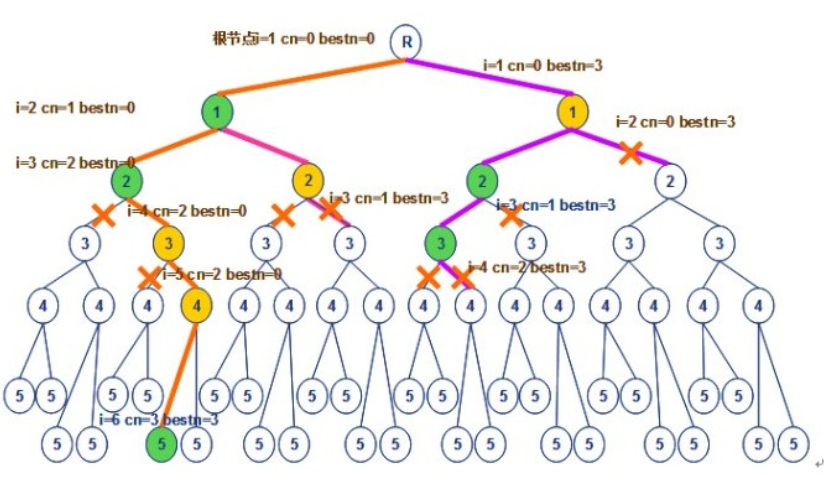
所以接下来我们的任务是首先判断这个点能否加入，若不能加入则kill掉左子树，然后在判断能否进入右子树，即还有没有足够多个点来使得右子树产生比已知最大团更大团，如果可以进入左子树，则还需要考虑加入该顶点和舍弃两种情况。这样执行剪枝策略可以有效提高搜索速度。那么我们怎么解决判断是否能够还有足够多个点使得右子树产生比已知最大团更大团？这里我们可以考虑执行一种特殊的扩展方式，我们先假设所有节点都是相连的，所以每次扩展时不是挑选节点，而是枚举节点，按照从小到大的顺序枚举节点，然后在判断是否相连，即从i=1开始，然后2,3,4，…,n这样对每一个节点进行判断，这样还有一个好处是相同情况的点会在数的同一层，这样我们就可以轻松知道假设节点Z在t层，总结点数为n个，已选的节点数为cn，那么剩余的除Z以外的节点数就是n-t个，若

那么就出发限界函数不进入此右子树，那么已选节点数可以通过每次选中某个节点时参数cn++来进行统计，自此核心判断公式就写完了，我们可以写出伪代码了。

# 二．伪代码：

1. **void** AdjacencyGraph::maxClique(**int** i){
2. //计算最大团的回溯代码
3. **if**(i>n){//到到达了叶子节点
4. //找到了一个最大团的元组解，更新
5. **for**(**int** j=1;j<=n;j++){
6. //记录这组解
7. bextx[j]=x[j];
8. //最大团的节点数就是此时解的节点数
9. bextn=cn;
10. **return**;//退出
11. }
12. }
13. //如果没到叶子节点则还是中间节点，首先判断能否进入左子树
14. //即顶点i是否与其他所有的已选顶点相连
15. **int** ok=1//默认是相连的
16. **for**(**int** j=1;j<i;j+=){
17. //j节点在已选最大团组里&&i和j不相连，找到一组即可
18. **if**(x[j]&&a[i][j]==NoEdge){
19. ok=0;//不相连
20. **break**;
21. }
22. }
23. **if**(ok)//符合全部相连
24. {
25. //将i加入最大团组中
26. x[i]=1;
27. //已选顶点数加一
28. cn++;
29. //检验i+1能否进入，递归调用
30. //并计算bestn
31. maxClique(i+1);
32. //很重要！！！
33. //还要回溯到i进入右子树检验不放回的情况
34. x[i]=0;
35. cn--;
36. }
37. //检验有没有必要进入右子树
38. //即除i节点以外选取全部剩余节点后能否形成更大团
39. **if**(cn+n-i>bextn){
40. //有必要进入&&不选i节点
41. x[i]=0;
42. //检验i+1能否进入，递归调用
43. //并计算bestn
44. maxClique(i+1)
45. }
46. }
47. //初始化和返还bestn
48. **int** AdjacencyGraph::MaxClique(**int** v[]){
49. x=**new** **int** [n+1]//存元组解
50. cn=0;//已选顶点数为0
51. bestn=0;//
52. bestx=v;
53. //从1开始注意检验顶点
54. maxClique(1);
55. **return** bestn;
56. }

例题讲解思路：



上图是对无向图G寻找最大团时状态空间树展开过程的图解，我们从R开始，逐一对顶点进行检验，首先R节点处相当于初始化，此时没有选点，cn=0,bestn=0。我们首先选节点1（因为默认单调递增检验顶点），然后进入1的左子树，检验2发现和1相连，可以加入，所以此时cn=2,bestn=0（记得吗，此时是a情况，虽然是一个完全子图，但是不是团，因为不是最大完全子图),然后检验节点3，发现3虽然和2相连，但是和1不相连，所以2不能展开左子树即不能选择3，此时判断能否进入右子树，此时已选节点数为2，剩余节点数为5-3=2（这里的图是显示3节点在第4层，但是代码由于默认从节点1位根节点出发，所以与这里的图略有出入，在代码里此时节点3就是在第三层），所以cn+n-i=4>bestn=0,即如果将4,5默认全选上还是有可能产生更大的团的，由于此时还没有形成团，所以bestn=0，所以可以对2进行右子树展开即放弃3，然后判断顶点4，发现4和2不相连，所以无法展开3的左子树即不能选择节点4，判断3能否右子树扩展，发现是可以的，所以判断5，发现5和1,2都相连，所以可以选5，此时对于4来说进行了左展开，此时到达了叶子节点，最大团产生了一个为{1,2,5},并且bestn=cn=3，然后注意对于可以加入的顶点，不要忘记还要回溯判断不选的情况，即回溯到了节点4，判断能够右子树扩展，发现此时cn=2(因为此时只选了1,2),出来5剩余的顶点数为5-5=0，所以cn+n-i=2bestn所以右展开，以此类推即完成了空间树的展开。

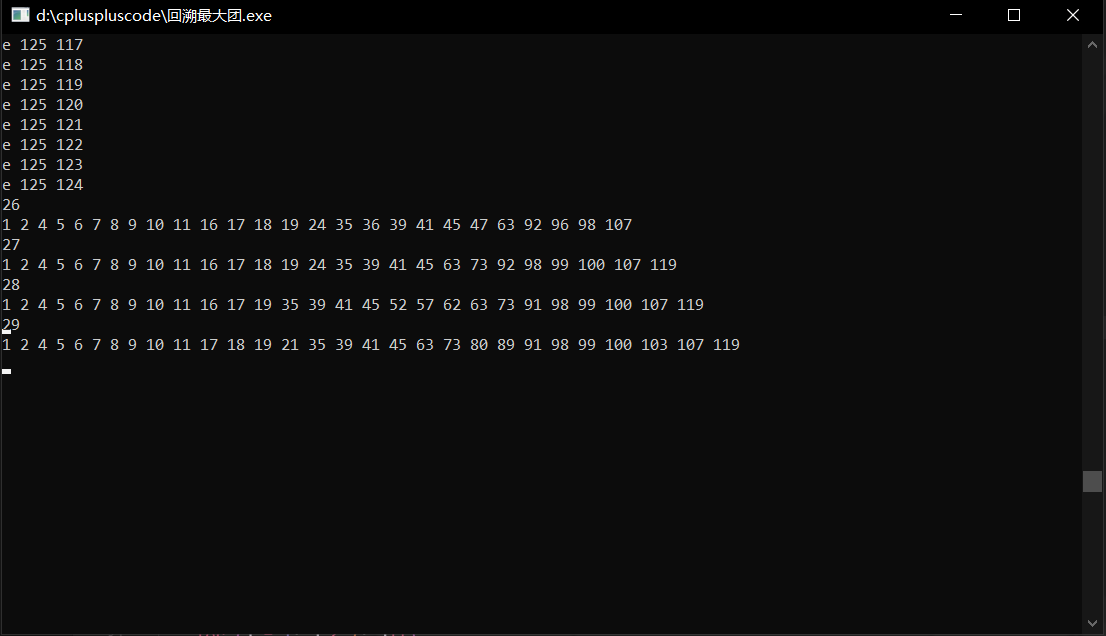
# 三、代码展示

1. #include <bits/stdc++.h>
2. **using** **namespace** std;
4. **const** **int** maxnum = 130;
5. **bool** a[maxnum][maxnum]; //图的邻接矩阵
6. **bool** x[maxnum];         //当前解
7. **int** cn;                 //当前团的顶点数
8. **int** bestn;              //当前的最优解
9. **int** n;                  //图G的顶点数
10. **int** e;                  //图G的边数
12. **void** traceback(**int** i)
13. {
14. **int** j;
15. **if** (i > n)
16. {
17. bestn = cn;
18. cout << bestn << endl;
19. **for** (j = 1; j <= n; j++)
20. {
21. **if** (x[j])
22. {
23. cout << j << " ";
24. }
25. }
26. cout << endl;
27. **return**;
28. }
30. **bool** ok = **true**;
31. **for** (j = 1; j < i; j++)
32. {
33. **if** (x[j] && !a[j][i]) //i与j不相连，左子树不展开
34. {
35. ok = **false**;
36. **break**;
37. }
38. }
39. **if** (ok) //进入左子树
40. {
41. cn++;
42. x[i] = **true**;
43. traceback(i + 1);
44. cn--; //为展开右子树做准备
45. }
46. **if** (cn + n - i > bestn) //剪枝策略
47. {
48. x[i] = **false**;
49. traceback(i + 1);
50. }
51. }
53. **int** main()
54. {
55. **int** i, u, v;
56. memset(a, **false**, **sizeof**(a));
57. memset(x, **false**, **sizeof**(x));
58. cin >> n >> e;
59. **char** c = 'e';
60. **for** (i = 0; i < e; i++)
61. {
62. cin >> c >> u >> v;
63. a[u][v] = **true**;
64. a[v][u] = **true**;
65. }
66. cn = bestn = 0;
67. traceback(1);
68. system("pause");
69. **return** 0;
70. }

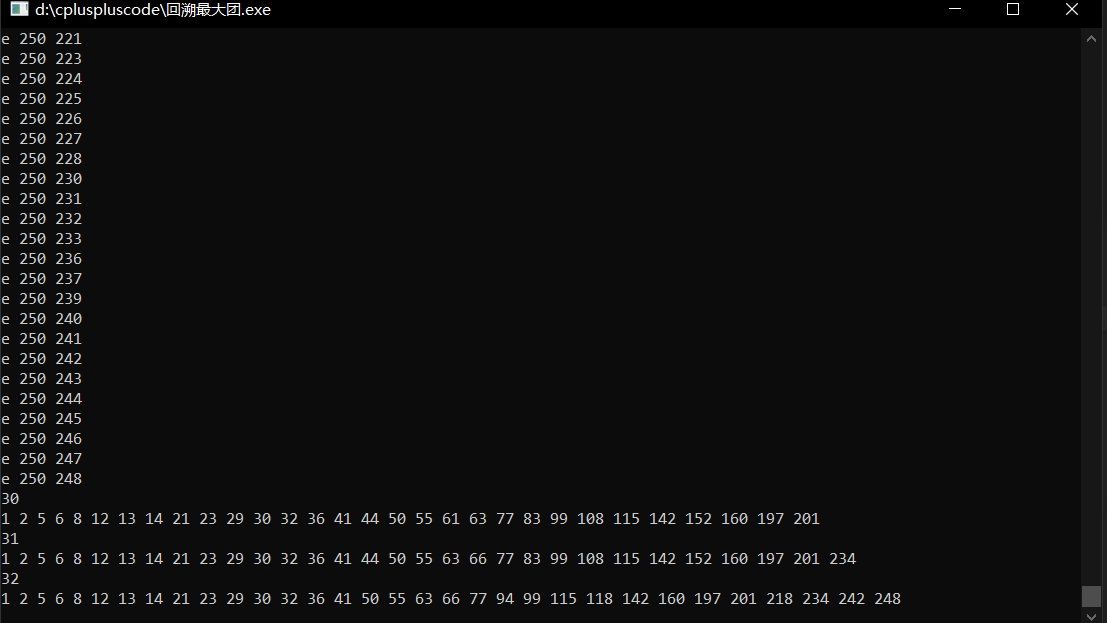
# 五、测试结果

所选数据集为DIMACS-benchmark, 37个数据集中的第一个和第二个数据集（具体的输入样例请查看文件夹内子团1.txt和子团2.txt），这里仅展示最终输出结果。

子团1.txt



子团2.txt



# 六、算法分析

时间复杂度为O（n^2）利用两个限界条件：、

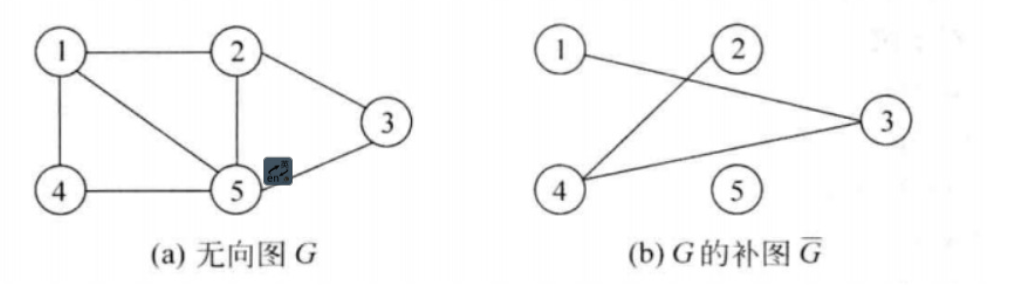
1. 展开左子树节点需要确保E-节点和已选节点全部相连
2. 展开右子树节点判断是否满足限界函数：cn+n-t>besttn

进行限界剪枝策略，相较于暴力展开状态空间树避免了大量的重复无用的操作，加快了搜索效率。

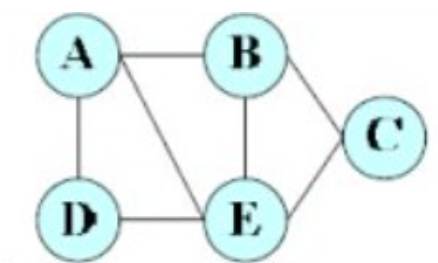
分枝限界法代码设计流程

# 一．设计思路

在分枝-限界法中，通常是使用广度优先搜索并且以最小消耗优先的方式解状态空间树。对于下图：



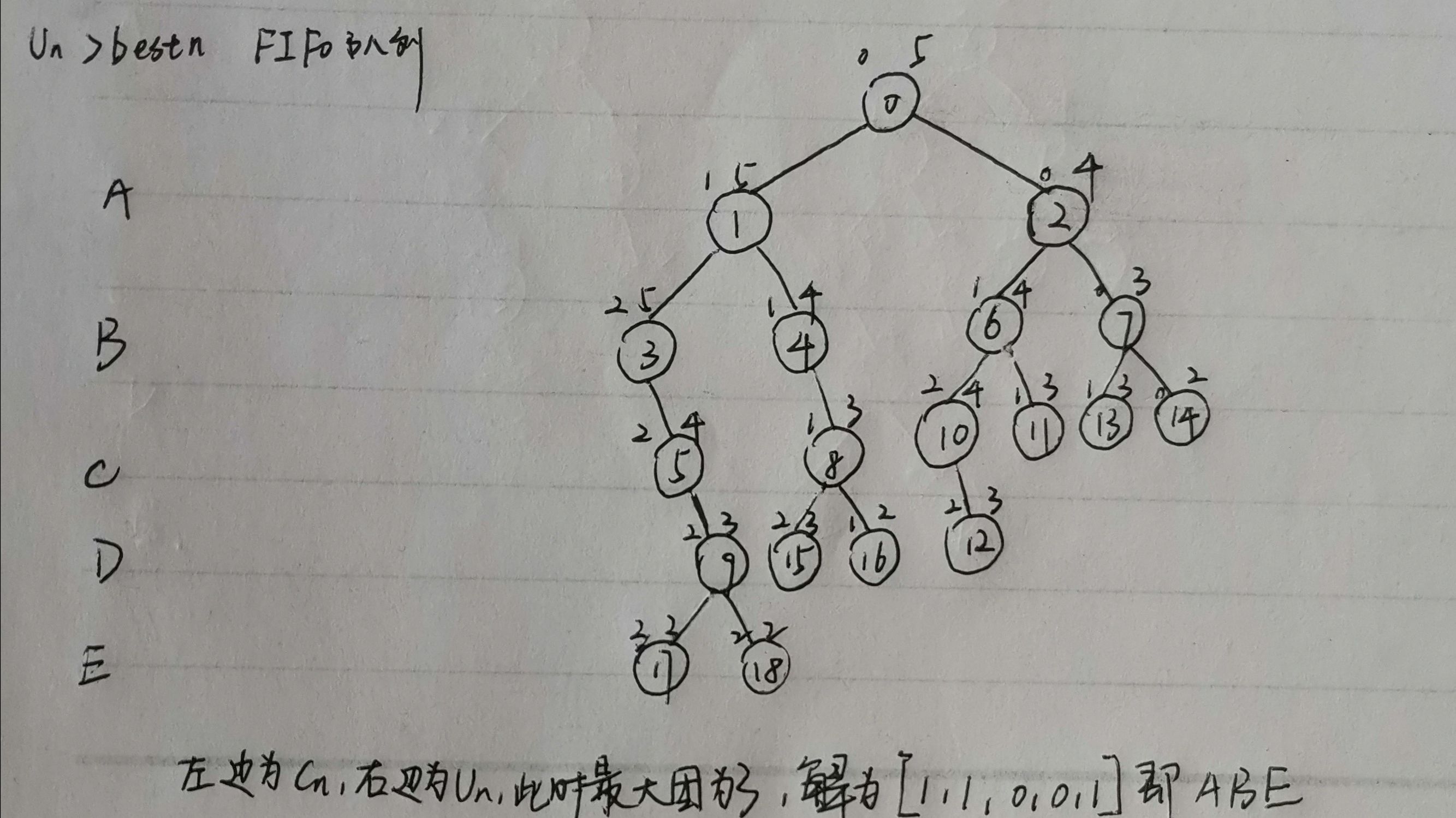
我们在使用分枝-限界法解决上面G的最大团问题之前，先考虑一下具体怎么实现，我们知道分枝-限界主要的限界方法就是设立几个条件，只有符合条件的节点才能进入活结点表，然后每次展开时都是展开活结点表的最优节点，这样不满条件即无法进入活结点表的节点也就被剪枝了，这就是分枝-限界的策略，所以我们需要寻找几个条件进行剪枝，首先我们可以很容易知道一个条件即想展开左子树需要左子节点和已知的团中所有节点相连，而对于所有节点只有满足cn+un>bestn(cn是当前已经选择的节点数，un是当前cn和n-t的和，n-t是剩余所有未选择节点的数，所以un即为可能产生的最大贪心效益团，bestn是当前已知的最大团的最优团数)，un=cn+n-t，所以只有满足已经选择的节点数和所有未选节点数之和比bestn大的才有可能继续产生更大团，所以只有满足cn+n-t的节点才能进入活结点表，接下来就是每次展开时都选择活结点表中的最优节点进行展开，我们知道对于0/1背包问题中我们每次选择的都是具有最大贪心效益值的节点进行展开，而对于最大团问题我们每次都选取活结点表中un最大的节点，但是难免会产生un相等的活结点表节点，此时根据不同的方法我们对状态空间树进行展开并对比两个策略的优劣。为了方便起见，我们将上图中G中的1,2,3,4,5用A,B,C,D,E代替：



二．两种策略展开空间树

（1）un>bestn&&FIFO队列

此时活结点表使用FIFO队列即先进先出，所以当面对un相同的节点，我们选择最先到的节点，当然节点还是要满足un从大到小排列的。那么状态空间树如下：



我们来一步步推导一下，其中括号包裹的是按顺序排列的活结点表，每次都选取队首节点（因为队首节点就是最优节点）,并且我们可以看出n-t表示的就是剩余节点数量且每一层都讨论的是某一个节点在不同情况下的选取情况，例如第一层表示的就是A的选取情况，第4层表示的就是D的选取情况。第零层根节点表示还没有选择节点，所以第零层的cn=0,un=5,此时还没有选择节点，所以已知最有团大小就是bestn=0。

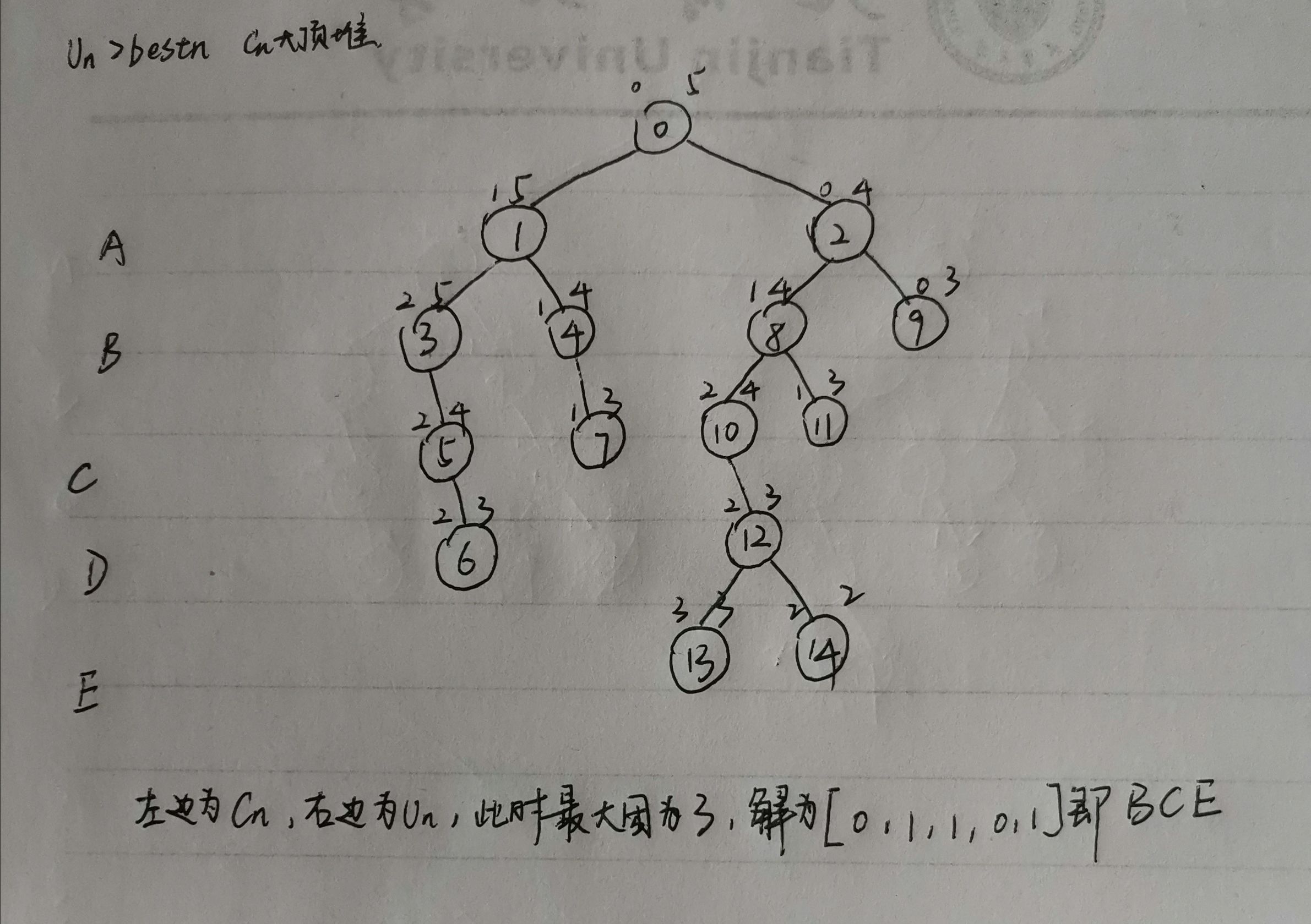
1. 首先根节点表示还没选择节点呢，所以bestn=0，此时左右分别讨论是否选择A得到两个节点1,2并且1此时的un=5不变，cn=1，2的un=4,cn=0，都满足cn>bestn,所以都插入节点表，并且此时更新bestn=1,虽然我们知道此时还没有形成团（但是可以特殊地看成是1个团大小为1的团），活结点表为（1,2）
2. 取1(此时1一定被展开成死节点，所以就不在活结点表中了），发现选A以后B可选可不选，所以同样左右展开的3和4,3的cn=2,un=5,4的cn=1,un=4,bestn=2,都可以插入活结点表，活结点表为(3,2,4)，虽然3比2来的晚，但是3的un更大，所以排在2前面，而4和2的un相同此时采取FIFO策略，2来的早所以2在前面。
3. 取3(此时3一定被展开成死节点，所以就不在活结点表中了），发现此时选取了A,B不能选C，因为C和A不相连，所以3只能右展开得5，此时cn=2,un=4满足un>bestn5加入活结点表，此时活结点表为(2,4,5)
4. 取2(此时2一定被展开成死节点，所以就不在活结点表中了），2左右展开均可以得节点节点6,7，6的cn=1,un=4,7的cn=0,un=3，均可以进活结点表，活结点表为(4,5,6,7)
5. 取4(此时4一定被展开成死节点，所以就不在活结点表中了），只能右展得节点8,cn=1,un=3，活结点表为(5,6,7,8)
6. 取5(此时5一定被展开成死节点，所以就不在活结点表中了），只能右展得9加入活结点表，活结点表为(6,7,8,9)
7. 取6(此时6一定被展开成死节点，所以就不在活结点表中了），6左右展均可以，得到10,11,10的cn=2,un=4,11的cn=1,un=3，所以加入活结点表，活结点表为(10,6,7,8,9,11),10在最前面是由于un最大。
8. 取7(此时7一定被展开成死节点，所以就不在活结点表中了），7左右展均可以，得到13和14,13的cn=1,un=3,14的cn=2,un=2，所以此时只有13加入活节点表，因为14的un=2==bestn=2，不大于所以剪枝不用再加入活结点表中讨论了。活结点表为(8,9,11,12,13)
9. 取8(此时8一定被展开成死节点，所以就不在活结点表中了），8左右展均可以，得到15,16,15的cn=2,un=3,16的cn=1,un=2,所以只有15加入活结点表，活结点表为(9,11,12,13,15)
10. 取9(此时9一定被展开成死节点，所以就不在活结点表中了），9左右展开均可以得到17和18，17的cn3,un=3,18的un=2,cn=2,此时发现已经到达叶节点了即当cn=un时就是叶子节点了，所以已经找到了最优解，及时在展开其他情况也只是得到不大于此时解的情况，所以此时的解就是一组最优解了，最大团数为bestn=cn=3,之所以不是18节点是因为17的情况更好，所以算法结束啦，最终就是最大团数为3，最优解为[1,1,0,0,1]即{A,B,E}，当然如果你继续展开要全部展开的话也可以，最终还会得到两组解为{B,C,E}和{A,D,E}但是并不会改变bestn的值了，所以剩下的活结点表节点销毁即可。

优化之处：

仔细观察，我们会发现在出现类似于2,5这种un相同的情况时采取的是FIFO策略存储，所以接下来是展开来的更早的节点，但是按照正常思维来说，我们考虑一下2和5,2节点表示此事还一个节点没选但是最好的可能情况为4，而节点5表示的是在已经选择了两个节点情况下最好的可能情况为4即5节点最好情况和2相同的情况下还保证了最差的情况为至少能得到大小为2的团，而节点2不排除得到大小为1或者0的团，所以按常理来说5比2更好，所以应该是展开5，所以在活结点表中我们应该将5放在2前面，所以我们采取一种大顶堆策略来实现活结点表存储，即每次都按照un从大到小排列的同时，当un相同时cn更大的排在前面。这样就得到了策略2

（2）un>bestn&&大顶堆

按照策略2进行展开，状态空间树如下：

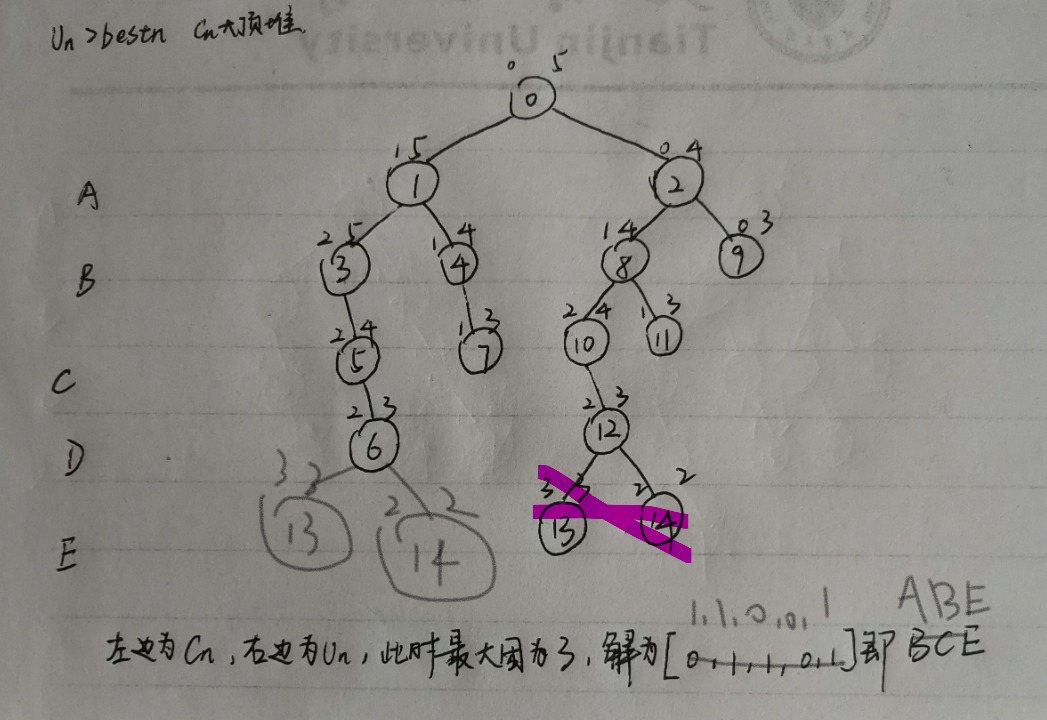


此时就是在un从大到小排序的同时，un相等的情况使用cn更大的节点排在前面的策略来存储，确实步数更少了所以也就优化了搜索效率，具体步骤如下：

1. 首先根节点表示还没选择节点呢，所以bestn=0，此时左右分别讨论是否选择A得到两个节点1,2并且1此时的un=5不变，cn=1，2的un=4,cn=0，都满足cn>bestn,所以都插入节点表，并且此时更新bestn=1,虽然我们知道此时还没有形成团（但是可以特殊地看成是1个团大小为1的团），活结点表为（1,2）
2. 取1(此时1一定被展开成死节点，所以就不在活结点表中了），发现选A以后B可选可不选，所以同样左右展开的3和4,3的cn=2,un=5,4的cn=1,un=4,bestn=2,都可以插入活结点表，活结点表为(3,2,4)，虽然3比2来的晚，但是3的un更大，所以排在2前面，而4和2的un相等，此时cn大的排在前面，所以活节点表为(3,4,2)
3. 取3(此时3一定被展开成死节点，所以就不在活结点表中了），只能右展得节点5,cn=2,un=4,所以5放入活节点表并且因为5比4更好，所以5直接插入到4前面，活节点表为(5,4,2)
4. 取5(此时5一定被展开成死节点，所以就不在活结点表中了），只能右展得到节点6，cn=2,un=3,所以放入活节点表中，活结点表为(4,2,6)
5. 取4(此时4一定被展开成死节点，所以就不在活结点表中了），只能右展得到节点7，cn=1,un=3，所以放入活结点表且7放在6的后面此时活结点表为(2,6,7)
6. 取2(此时2一定被展开成死节点，所以就不在活结点表中了），左右均可展开得到8,9，8的cn=1，un=4,9的cn=0,un=3,所以8放入活结点表头因为他的un最大，而9在最末端因为un=3且cn=0最小，所以活结点表为(8,6,7,9)
7. 取8(此时8一定被展开成死节点，所以就不在活结点表中了），8左右均可展开，所以得到10和11,10的cn=2,un=4,11的cn=1,un=3,所以10放在最前面因为un最大，而11放在7前面，我们发现此时出现了更为特殊地情况即11和7的cn和un均相等，此时可以采取两种策略，第一种就是FIFO，先到先出，这样可以，但是还有一种策略就是将11插入到7前面因为既然11和7的情况完全相同那么我们就继续选择最近完成的情况11进行展开。活结点表为(10,6,11,7,9)
8. 取10(此时10一定被展开成死节点，所以就不在活结点表中了），10只能右展得到12，此时发现又出现特殊情况12和6的情况完全相同，我们还是采取不切换的策略所以活结点表为(12,6,11,7,9)
9. 取12(此时12一定被展开成死节点，所以就不在活结点表中了），左右展开得到13和14并且此时13和14是cn=un所以到达了叶子节点，所以13就是最优解，最大团数为3,解为[0,1,1,0,1]即为{B,C,E},此时也是继续展开肯能还会得到其他团为3的解，但是总体看来最大团数不会再发生改变就是3，所以活结点表中剩余节点销毁即可，算法结束。

思考：对于cn和un都相等的情况使用FIFO是否更好？

事实是使用FIFO策略更好，因为这样好实现功能，所以我们对上面的进行小更改，此时按照FIFO策略状态空间树展开如下：



三．代码展示

我们用的是FIFO队列实现实现简便

1. #include <bits/stdc++.h>
3. **using** **namespace** std;
5. #define MAXN 130     //邻接矩阵值
6. #define INF 0x3f3f3f //无限大值
8. **int** m[MAXN][MAXN]; //邻接矩阵
10. **int** bestN = 0; //已知的当前最大团数
12. **struct** Node
13. {
14. **int** level;   //层数
15. **int** cn;      //当前的已选节点数量
16. **int** upbound; //un,就是最理想情况团的大小
17. Node \*parent;
18. **int** isLeft;                                                         //是否是左展开节点
19. Node(**int** \_level, **int** \_cn, **int** \_upbound, Node \*\_parent, **int** \_isLeft) //初始化赋值函数
20. {
21. level = \_level;     //节点层数
22. cn = \_cn;           //当前已选择的节点数
23. upbound = \_upbound; //上界，即产生的最大团数
24. parent = \_parent;   //父节点
25. isLeft = \_isLeft;
26. }
27. };
29. **struct** cmp
30. {
31. **bool** operator()(Node \*n1, Node \*n2)
32. {
33. **return** n1->upbound < n2->upbound; // 规定比较规则为比较上界，制定活结点表排列规则
34. }
35. };
37. priority\_queue<Node \*, vector<Node \*>, cmp> pq; //初始化活结点表，权重是最大贪心团数值
39. **bool** constrain(Node \*cNode) //用来判断左展开节点是否和已选的节点全部相连
40. {
41. **int** tmp[MAXN];
42. **int** n = cNode->level;
43. tmp[n] = 1;
44. **while** (cNode->parent != NULL)
45. {
46. tmp[cNode->level - 1] = cNode->isLeft ? 1 : 0;
47. cNode = cNode->parent;
48. }
50. **for** (**int** i = 1; i <= n; i++)
51. {
52. **if** (tmp[i])
53. {
54. **for** (**int** j = 1; j <= n; j++)
55. {
56. **if** (i == j)
57. **continue**;
58. **if** (tmp[j] && !m[i][j])
59. **return** **false**;
60. }
61. }
62. }
64. **return** **true**;
65. }
67. **void** maxGroupPrior(**int** n)
68. {
69. // 初始化
70. Node \*root = **new** Node(1, 0, n, NULL, **false**);
71. pq.push(root);
73. Node \*bestNode = NULL;
75. // 开始广度搜索
76. **while** (!pq.empty())
77. {
78. // 队首
79. Node \*cNode = pq.top();
80. pq.pop(); //每次都取队首元素
82. **if** (cNode->level == n + 1)
83. {
84. bestN = cNode->cn;
85. bestNode = cNode;
86. **break**;
87. }
89. // 左判约束
90. **if** (constrain(cNode))
91. {
93. // 新建可行节点
94. Node \*tmp1 = **new** Node(cNode->level + 1, cNode->cn + 1, n - cNode->level + cNode->cn + 1, cNode, **true**);
96. **if** (cNode->cn + 1 > bestN)
97. {
98. bestN = cNode->cn + 1;
99. bestNode = tmp1;
100. }
101. // 将该节点加入优先队列
102. pq.push(tmp1);
103. }
105. // 右判界
106. **if** (cNode->upbound > bestN)
107. {
108. // 新建可行节点
109. Node \*tmp2 = **new** Node(cNode->level + 1, cNode->cn, n - cNode->level + cNode->cn, cNode, **false**);
111. **if** (cNode->cn > bestN) //是否满足un>bestn
112. {
113. bestN = cNode->cn;
114. bestNode = tmp2;
115. }
116. // 将该节点加入优先队列
117. pq.push(tmp2);
118. }
119. }
121. // 输出最优解
122. cout << bestN << endl;
123. **while** (bestNode->parent != NULL)
124. {
125. **if** (bestNode->isLeft)
126. {
127. cout << 1 << " ";
128. }
129. **else**
130. {
131. cout << 0 << " ";
132. }
133. bestNode = bestNode->parent;
134. }
135. }
137. **int** main()
138. {
139. **int** n, e;
140. cin >> n >> e;
141. //初始化邻接矩阵为零矩阵
142. memset(m, 0, **sizeof**(m));
143. **for** (**int** i = 0; i < e; i++)
144. {
145. **int** index1, index2;
146. **char** c = 'e';
147. cin >> c >> index1 >> index2;
148. m[index1][index2] = 1; //赋值边
149. m[index2][index1] = 1;
150. }
151. maxGroupPrior(n);
152. system("pause");
153. **return** 0;
154. }

# 四、结果分析

同样使用限界条件剪枝策略：

1. 判断左子树节点能否展开加入活结点表的条件是与已选所有的节点全部相连
2. 判断右子树节点能否展开加入活结点表的条件是cn+n-t>bestn

同时根据活结点表的不同优化效率也有所差异，但是相较于bfs广度搜索枚举效率更高，准确度无差别。

回溯法和分支限界法的区别

我们了解到回溯法实际上就是在dfs基础上加以限界函数来加快搜索速度，其中无论是哪种题型，其核心思想都是一样的，设置元组解，对利用限界条件对状态空间树展开加以剪枝，其中，我们都是在能否展开，能够左子树展开，能否右子树展开这三处关键点进行限界条件添加，当然如果特别复杂的限界判断可以适当不加，一定要注意对于这种dfs回溯永远都是能左子树展开，优先一直展开左子树，即E-节点优先一直向下延伸，同时对于左子树展开的同时也要回溯进行右子树展开，这样问题才能讨论完全，直至每次到达叶子节点得到一组可行解，记住要更新限界条件等，然后在所有可行接中最终得到最优解，回溯法就是每次都尽量走能取得更好情况的左展（这里有一个左展限界），用时也要进行回溯讨论能否右展（右展限界）最终是得到许多可能产生最优情况的解再对比这多组解得到最优解。而分枝-限界就是一次性左右展都展开（如果有特别需要左右展有限界条件）同时对于每一个节点（无论是左展还是右展）都要用一个限界条件来判断是否可以加入活结点表，只有能够满足条件进入活结点表的节点才有被展开的资格，然后每次展开时都是选取活结点表队首元素即当下能产生最好情况的节点进行展开，最终得到的解就是最优解。无优劣之分，不一定每次分枝-限界都比回溯法搜索效率高。但是相较于深搜和广搜，这两种方法对于较大的量级输入样例能够更加快速高效的找到最优解。

文献参考

1. <https://www.cnblogs.com/wkfvawl/p/11923848.html>
2. <https://blog.csdn.net/shimin520shimin/article/details/48968127>

《算法导论》回溯法和分支限界法章节与最大团问题案例分析