# 放牧

创建前缀和（sum[i][j]代表从1,1到i,j的“1”的数量）

sum[i][j]=

sum[i-1][j]+sum[i][j-1]- sum[i-1][j-1]sum[i][j]=sum[i−1][j]+sum[i][j−1]−sum[i−1][j−1]

# 防守

【题目描述】3.28 通信服务（comm）POJ 3659

通信公司需要在N个地区建立通信网络，已知每在一个地区建立一个信号塔，就可以对它自身及周围相邻的地区提供通信服务，恰好N－1个地区是相邻的，试问如何建最少数量的信号塔来保证所有地区的通信服务畅通？

【输入格式】

第一行为一个整数N（1≤N≤10 000)）。

随后N－1行，每行指定了一对相邻的地区A和B (1≤A≤N；1≤B≤N；A≠B)。

【输出格式】

输出一行一个数字，即信号塔建造的最少数。

【输入样例】

5

1 3

5 2

4 3

3 5

【输出样例】

2

【算法分析】

本题可以使用树形DP，也可以用贪心法（即选择父结点建立信号塔比选择子结点建立信号塔更划算）求树的最小支配集算法来做。所谓树的最小支配集，是指从所有结点中取尽量少的点组成一个集合，使得剩下的所有点都与取出来的点有边相连，结点个数最少的支配集被称为最小支配集。其算法过程为：

（1）以任意点例如1号点深度优先搜索即DFS整棵树，求出每个点在DFS中的编号（存入代码中的DfsNode[ ]）和每个点的父结点编号（存入代码中的father[ ]）。

（2）按DFS的反向序列检查，如果当前点既不属于支配集也不与支配集中的点相连，且它的父结点也不属于支配集，将其父结点加入支配集Set[ ]（选择父结点肯定比选择当前结点更优），支配集个数加1。为什么要按照DFS的反向进行检查呢？因为这样可以保证对于每个点来说，当其子树都被处理过后才轮到该结点的处理，保证了贪心的正确性。如图3.63所示，正向DFS时，如果每次优先选父结点，这种方式选出的支配集是5个，显然是错误的。

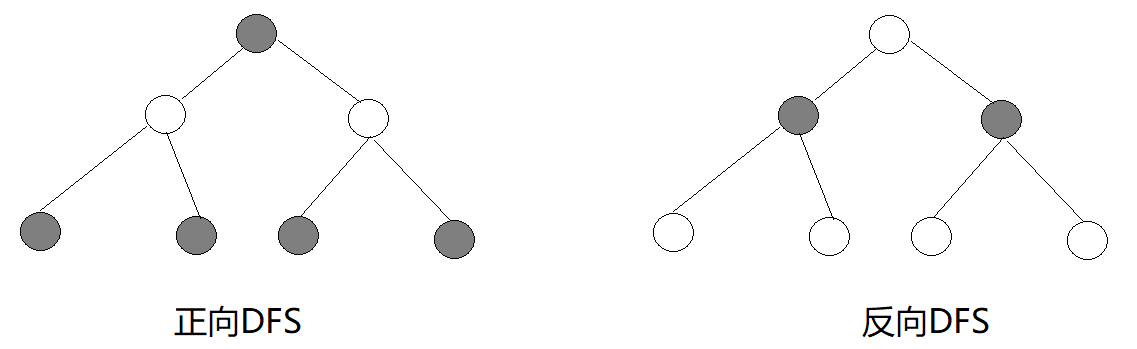


图3.63

（3）用s[ ]标记当前结点、当前结点的父结点(加入支配集)、当前结点的父结点的父结点(与支配集中的点相连)被覆盖。

此题可以使用STL模板中的vector模拟链表的方式实现树的存储，即定义一个vector数组edg[N]，某个二叉树结构保存到edg[N]的样例如图3.64所示，这种存储方式可以很方便的枚举出与某个结点连接的其他所有结点。

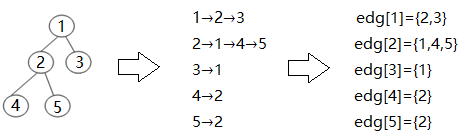


图3.64

参考程序如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | //通信服务  #include <bits/stdc++.h>  **using** **namespace** std**;**  const int N**=**2e4**+**10**;** //2e4即20000  int n**,**tim**=**0**;**  int DfsNode**[**N**],**Father**[**N**];**  vector**<**int**>**edg**[**N**];**  bool s**[**N**],**Set**[**N**];** //s[]为覆盖点,Set[]为支配集，注意Set不能小写  void Dfs**(**int u**,**int father**)**  **{**  DfsNode**[**tim**++]=**u**;** //将u点编号为tim存入DfsNode  **for(**int i**=**0**;** i**<**edg**[**u**].**size**();** i**++)**//继续DFS结点u的子结点  **{**  int v**=**edg**[**u**][**i**];** //枚举到u的子结点v  **if(**v**!=**father**)**  **{**  Father**[**v**]=**u**;** //标记结点v的父结点为u  Dfs**(**v**,**u**);** //继续DFS  **}**  **}**  **}**  int Greedy**()**  **{**  int ans**=**0**;**  **for(**int i**=**n**-**1**;** i**>=**0**;** i**--)** //反序DFS  **{**  int t**=**DfsNode**[**i**];**  **if(!**s**[**t**])**//当前点未被覆盖，既它不属于支配集，也不与支配集中的点相连  **{**  **if(!**Set**[**Father**[**t**]])** //当前点的父结点不属于支配集  **{**  Set**[**Father**[**t**]]=true;** //将父结点加入支配集  ans**++;** //支配集数目加一  **}**  s**[**t**]=true;** //标记当前结点被覆盖  s**[**Father**[**t**]]=true;** //标记当前结点的父结点被覆盖  s**[**Father**[**Father**[**t**]]]=true;** //标记当前结点的父结点的父结点被覆盖  **}**  **}**  **return** ans**;**  **}**  int main**()**  **{**  scanf**(**"%d"**,&**n**);**  **for(**int i**=**1**,**u**,**v**;** i**<**n**;** i**++)**  **{**  scanf**(**"%d%d"**,&**u**,&**v**);**  edg**[**u**].**push\_back**(**v**);**  edg**[**v**].**push\_back**(**u**);**  **}**  Dfs**(**1**,**0**);**  printf**(**"%d\n"**,**Greedy**());**  **return** 0**;**  **}** |

树形动规的做法是：随机把一个点看做树的根结点，设：

（1）dp[i][0]表示选结点i，并且覆盖了以i为根的子树所有点的最小结点数；

（2）dp[i][1]表示不选结点i，但选了i的子结点（至少1个）的情况下，覆盖了以i为根的子树所有点的最小结点数；

（3）dp[i][2]表示不选结点i，i也没有被其子结点覆盖的情况下，覆盖了以i为根的子树所有点的最小结点数。（注：即i将要被它的父结点覆盖）

例如从结点1开始DFS深搜，则最终答案应为min(dp[1][0]，dp[1][1])，显然这是一个树上01背包问题，即结点1选或不选（不选则必有子结点被选）的最小值。

以图3.65为例来说明，结点u的父结点为fa结点，子结点Vi可能有多个：

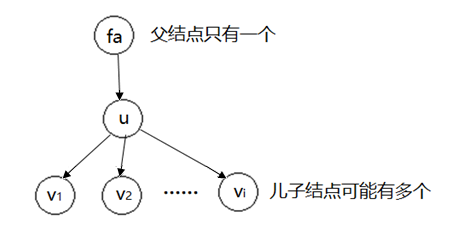


图3.65

当选结点u自身时，得dp[u][0]＝1+∑min(dp[vj][0]，dp[vj][1]，dp[vj][2])，其中1≤j≤i；这是因为u点已被选，所以它的子结点vj可选可不选，故选择每个子结点的三种状态的最小值累加即可。

当不选择结点u自身，但u结点被父结点覆盖时，得dp[u][2]＝∑min(dp[v][0]，dp[v][1])；这是因为u点已经被父结点覆盖，所以它对子结点v的选择只需是dp[v][0]和dp[v][1]中的最小值即可。

当不选择结点u自身，选择它的子结点时，由dp[u][0]定义，u的子结点必须选一个，当然选择多个子结点也是可以的。分以下几种情况讨论：

（1）如果u没有子结点，则dp[u][1]＝∞；

（2）如果选择了子结点，则dp[u][1]＝∑min(dp[v][0]，dp[v][1])；

（3）如果没有选择子结点（因为dp[v][0]＞dp[v][1]），就必须强制选择一个，即dp[u][1] ＝∑min(dp[v][0]，dp[v][1])＋min(dp[v][0]－dp[v][1])。（注：＋min(dp[v][0]－dp[v][1])为补差值）

动规参考程序如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | //通信服务 — DP  #include <bits/stdc++.h>  **using** **namespace** std**;**  const int MAXN **=** 10010**;**  const int INF **=** 0x3f3f3f3f**;**  vector **<**int**>** Edg**[**MAXN**];**  int dp**[**MAXN**][**MAXN**];**  int vis**[**MAXN**];**  void Dfs**(**int u**)**  **{**  vis**[**u**]=**1**;** //标记已访问  int flag**=**1**,**tmp**=**INF**;**  **for(**int i**=**0**;** i**<**Edg**[**u**].**size**();** i**++)** //枚举子结点  **{**  int v**=**Edg**[**u**][**i**];**  **if(!**vis**[**v**])** //如果子结点没标记已访问  **{**  Dfs**(**v**);**  dp**[**u**][**0**]+=**min**(**dp**[**v**][**0**],**min**(**dp**[**v**][**1**],**dp**[**v**][**2**]));** //(1)取u结点  dp**[**u**][**2**]+=**min**(**dp**[**v**][**0**],**dp**[**v**][**1**]);**//(3)不取u结点，也没被子结点覆盖  **if(**dp**[**v**][**0**]<=**dp**[**v**][**1**])** //(2)至少选一个子结点  **{**  dp**[**u**][**1**]+=**dp**[**v**][**0**];**  flag**=**0**;** //选了一个子结点就要做标记  **}**  **else** //如没取子结点，维护dp[v][0]-dp[v][1]  **{**  dp**[**u**][**1**]+=**dp**[**v**][**1**];**  tmp**=**min**(**tmp**,**dp**[**v**][**0**]-**dp**[**v**][**1**]);**//tmp表示它们差值最小的那个差值  **}**  **}**  **}**  **if(**flag**)** //未选子结点,则必须强制换一个子结点加入  dp**[**u**][**1**]+=**tmp**;** //加上这个差值  **return** **;**  **}**  int main**()**  **{**  int n**;**  scanf**(**"%d"**,&**n**);**  int u**,**v**;**  **for(**int i**=**1**;** i**<**n**;** i**++)**  **{**  scanf**(**"%d%d"**,&**u**,&**v**);**  Edg**[**u**].**push\_back**(**v**);**  Edg**[**v**].**push\_back**(**u**);**  **}**  **for(**int i**=**0**;** i**<=**n**;** i**++)** //dp[i][0]的i从0开始，因为存在下标为0的结点  **{**  dp**[**i**][**0**]=**1**;**  dp**[**i**][**1**]=**0**;**  dp**[**i**][**2**]=**0**;**  **}**  **if(**n**==**1**)**  printf**(**"1\n"**);**  **else**  **{**  Dfs**(**1**);**  printf**(**"%d\n"**,**min**(**dp**[**1**][**0**],**dp**[**1**][**1**]));**//树上01背包,选或不选  **}**  **return** 0**;**  **}** |

回到本题：

当我们分析一个节点的时候，很容易想到起码有两种情况：在这个节点设或者不设。但是如果单纯地只是分析它不设，那么依据题意，它必须得被观测到，那么它的子节点到底要不要设呢？此时便又要分两种情况：被儿子观测或是被父亲观测。因此抽象成了三种状态转移：

每个结点有三种情况，分别设为：

dp[i][0]:自己守卫

dp[i][1]:被儿子守

dp[i][2]:被父亲守

然后对于每种状态分析转移方程:

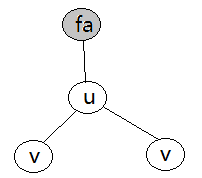
dp[i][0]:因为自己守了，所以儿子怎么样的状态都行。

dp[u][0]+=min(dp[v][2], dp[v][0],dp[v][1]);

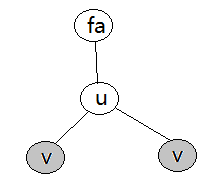
dp[i][2]:被父亲守，对于每个儿子来说，它既可以自己守也可以被儿子的儿子守。

dp[u][2]+=min(dp[v][1],dp[v][0]);

为什么这时候不再需要被父节点守呢？这是因为如果父节点也守那么这个点就不再需要被它的父节点守了。



dp[i][1]:被儿子守，dp[u][1]=min(dp[u][1],dp[u][2]-min(dp[v][1],dp[v][0])+dp[v][0]



如果点u被自己的儿子守，那么就意味着u的所有子节点中至少有一个点需要自己守自己。那么什么情况下u的子节点不会选择守呢？对于子节点v来说，自己守的代价为dp[v][0],而因为它的父节点u不会守自己，所以v还有一种选择，也就是被自己的儿子守，即dp[v][1];所以当dp[v][0]>dp[v][1]时，v不会守。

· 同时需要注意此时的dp[u][2]表示的是所有子节点的最小总价值，dp[u][2]-min(dp[v][1],dp[v][0])+dp[v][0] 表示的就是如果u点被儿子守，所有儿子中至少有一个点守的价值。然后在其中取最小值。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | //防守  #include <bits/stdc++.h>  **using** **namespace** std**;**  const int MAXN**=**2000**;**  struct node  **{**  int to**,** next**;**  **}** edge**[**MAXN**];**  int n**,**k**,**cnt**=**1**,**ans**;**  int head**[**MAXN**],**val**[**MAXN**],**in**[**MAXN**],**dp**[**MAXN**][**3**];**  void Addedge**(**int from**,**int to**)**  **{**  edge**[**cnt**].**to**=**to**;**  edge**[**cnt**].**next**=**head**[**from**];**  head**[**from**]=**cnt**++;**  **}**  void Dfs**(**int u**)**  **{**  dp**[**u**][**0**]=**val**[**u**];**  dp**[**u**][**1**]=**1**<<**30**;**  **for(**int i**=**head**[**u**];** i**!=-**1**;** i**=**edge**[**i**].**next**)**  **{**  int v**=**edge**[**i**].**to**;**  Dfs**(**v**);**  dp**[**u**][**0**]+=**min**(**dp**[**v**][**2**],**min**(**dp**[**v**][**0**],**dp**[**v**][**1**]));**//自己守,三种情况都有  dp**[**u**][**2**]+=**min**(**dp**[**v**][**1**],**dp**[**v**][**0**]);** //父亲守, 则可由它儿子守也可自己守  **}**  **for(**int i**=**head**[**u**];** i**!=-**1**;** i**=**edge**[**i**].**next**)**//儿子守  **{**  int v**=**edge**[**i**].**to**;**  dp**[**u**][**1**]=**min**(**dp**[**u**][**1**],**dp**[**u**][**2**]-**min**(**dp**[**v**][**1**],**dp**[**v**][**0**])+**dp**[**v**][**0**]);**  **}**  **}**  int main**()**  **{**  memset**(**head**,-**1**,sizeof(**head**));**//结点有0，所以要初始为-1  scanf**(**"%d"**,&**n**);**  **for(**int t**=**n**,**node**,**m**;** t**;** t**--)**  **{**  scanf**(**"%d"**,&**node**);**  scanf**(**"%d%d"**,&**val**[**node**],&**m**);**  **for(**int i**=**1**,**son**;** i**<=**m**;** i**++)**  **{**  scanf**(**"%d"**,&**son**);**  Addedge**(**node**,**son**);**  in**[**son**]++;**  **}**  **}**  **for(**int i**=**1**;** i**<=**n**;** i**++)**  **if(!**in**[**i**])** //找到根结点  **{**  Dfs**(**i**);**  printf**(**"%d\n"**,**min**(**dp**[**i**][**0**],**dp**[**i**][**1**]));**  **return** 0**;**  **}**  **}** |

# 极北之雪

树的直径

【题目描述】3.33 极北之地（road）POJ 2631

在极北之地建设和维护道路是一件非常困难的事情，因此那里任意两个村庄之间只有一条道路连通（双向）且所有村庄（多达10 000个，从1开始编号）都可以直接或间接到达。

你的工作是计算出两个最远村庄的道路长度。

【输入格式】

输入有多行，每行有三个整数，即两个村庄的编号及之间的道路长度。

【输出格式】

输出两个最远村庄的道路长度。

【输入样例】

5 1 6

1 4 5

6 3 9

2 6 8

6 1 7

【输出样例】

22

【算法分析】

样例如图3.69所示，其中粗线即为所求的最长道路。

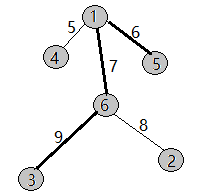


图3.69

显然题目是求树中最远的两个结点之间的距离，即树的直径（也可以称作树的最长链）。解决办法是第一次任意选一个点进行DFS（或BFS）找到离它最远的点，此点就是最长链的一个端点，再以此点进行DFS（或BFS），找到离它最远的点，此点就是最长链的另一个端点。

|  |  |
| --- | --- |
|  | 假设结点s和t是最长链的两个端点，任选一结点u进行DFS搜到的结点为v，可能情况如下：  （1）v点在最长链上，那么Dist[u，v]＞Dist[u，v]＋Dist[v，s]，与已知矛盾。  （2）v点不在最长链上，则在最长链上选择一个结点x，则Dist[u，v]＞Dist[u，x]＋Dist[x，t]，那么有Dist[s，v]＝Dist[s，x]＋Dist[x，u]＋Dist[u，v]＞Dist[s，x]＋Dist[x，t]＝Dist[s，t]，即Dist[s，v]＞Dist[s，t]，与已知矛盾。  （3）其他可能的情况均可以此反证法证明。 |

基于DFS的参考程序如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | //极北之地 — DFS算法  #include<bits/stdc++.h>  **using** **namespace** std**;**  const int MAXN**=**100005**;**  int head**[**MAXN**],**visit**[**MAXN**],**dist**[**MAXN**];**  int node**=**1**,**ans**,**k**;**  struct Edge  **{**  int v**,**len**;**  int next**;**  **}** edge**[**MAXN**<<**2**];**  void AddEdge**(**int u**,**int v**,**int l**)** //使用前向星表示法，请参见图的章节  **{**  edge**[**k**].**v**=**v**;**  edge**[**k**].**len**=**l**;**  edge**[**k**].**next**=**head**[**u**];**  head**[**u**]=**k**++;**  **}**  void Dfs**(**int u**,**int lenth**)**  **{**  visit**[**u**]=**1**;**  **for(**int i**=**head**[**u**];** i**!=-**1**;** i**=**edge**[**i**].**next**)**  **{**  int v**=**edge**[**i**].**v**;**  **if(!**visit**[**v**])**  **{**  visit**[**v**]=**1**;**  dist**[**v**]=**lenth**+**edge**[**i**].**len**;**  **if(**dist**[**v**]>**ans**)**  **{**  ans**=**dist**[**v**];**  node**=**v**;**  **}**  Dfs**(**v**,**dist**[**v**]);**  **}**  **}**  **}**  int main**()**  **{**  int l**,**r**,**len**;**  memset**(**head**,-**1**,sizeof(**head**));**  **while(~**scanf**(**"%d%d%d"**,&**l**,&**r**,&**len**))**  **{**  AddEdge**(**l**,**r**,**len**);**  AddEdge**(**r**,**l**,**len**);**  **}**  **for(**int i**=**1**;** i**<=**2**;** i**++)**  **{**  memset**(**visit**,**0**,sizeof(**visit**));**  ans**=**0**;**  Dfs**(**node**,**0**);** //第一次node为1，第二次node为最长链的一个端点  **}**  printf**(**"%d\n"**,**ans**);**  **return** 0**;**  **}** |

基于BFS的参考程序如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | //极北之地 — BFS算法  #include<bits/stdc++.h>  **using** **namespace** std**;**  const int MAXN**=**100005**;**  int head**[**MAXN**],**visit**[**MAXN**],**dist**[**MAXN**];**  int node**=**1**,**ans**,**k**;**  struct Edge  **{**  int v**,**len**;**  int next**;**  **}** edge**[**MAXN**<<**2**];**  void AddEdge**(**int u**,**int v**,**int l**)** //使用前向星表示法，请参见第五章  **{**  edge**[**k**].**v**=**v**;**  edge**[**k**].**len**=**l**;**  edge**[**k**].**next**=**head**[**u**];**  head**[**u**]=**k**++;**  **}**  void Bfs**(**int p**)**  **{**  queue**<**int**>**q**;**  visit**[**p**]=**1**;**  q**.**push**(**p**);**  **while(!**q**.**empty**())**  **{**  int u**=**q**.**front**();**  q**.**pop**();**  **for(**int i**=**head**[**u**];** i**!=-**1**;** i**=**edge**[**i**].**next**)**  **{**  int v**=**edge**[**i**].**v**;**  **if(**visit**[**v**]==**0**)**  **{**  dist**[**v**]=**dist**[**u**]+**edge**[**i**].**len**;**  visit**[**v**]=**1**;**  q**.**push**(**v**);**  **if(**dist**[**v**]>**ans**)**  **{**  ans**=**dist**[**v**];**  node**=**v**;**  **}**  **}**  **}**  **}**  **}**  int main**()**  **{**  int l**,**r**,**len**;**  memset**(**head**,-**1**,sizeof(**head**));**  **while(~**scanf**(**"%d%d%d"**,&**l**,&**r**,&**len**))**  **{**  AddEdge**(**l**,**r**,**len**);**  AddEdge**(**r**,**l**,**len**);**  **}**  **for(**int i**=**1**;** i**<=**2**;** i**++)**  **{**  memset**(**visit**,**0**,sizeof(**visit**));**  memset**(**dist**,**0**,sizeof(**dist**));**  ans**=**0**;**  Bfs**(**node**);**  **}**  printf**(**"%d\n"**,**ans**);**  **return** 0**;**  **}** |

BFS及DFS算法的时间复杂度均为O(n)，其优点是可以在第一次DFS/BFS时记录前驱，缺点是代码量稍大。

计算树的直径还可以使用树形动规算法，其时间复杂度为O(n)，其优点是代码量少，缺点是不容易记录路径。

设first[i]表示以结点i为根的子树中，i到叶子结点距离的最大值，second[i]表示以结点i为根的子树中，i 到叶子结点距离的次大值。

因为树的直径必然是树上某一个点开始往下的最长链和次长链之和，所以答案即为：max{first[i]＋second[i]}。

参考程序如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | //极北之地 — DP  #include<bits/stdc++.h>  **using** **namespace** std**;**  const int MAXN**=**10010**;**  struct edge  **{**  int v**,**len**,**next**;**  **}** edge**[**MAXN**<<**2**];**  int ans**,**k**;**  int head**[**MAXN**],**first**[**MAXN**],**second**[**MAXN**];** //前向星，最长路，次长路  void AddEdge**(**int u**,**int v**,**int len**)** //前向星表示法参见第五章  **{**  edge**[++**k**].**v**=**v**;**  edge**[**k**].**len**=**len**;**  edge**[**k**].**next**=**head**[**u**];**  head**[**u**]=**k**;**  **}**  void Dp**(**int u**,**int father**)** //树形动规  **{**  **for(**int i**=**head**[**u**];** i**;** i**=**edge**[**i**].**next**)**  **{**  int v**=**edge**[**i**].**v**;**  **if(**v**!=**father**)**  **{**  Dp**(**v**,**u**);**  **if(**first**[**v**]+**edge**[**i**].**len**>**first**[**u**])** //能更新最大值  **{**  second**[**u**]=**first**[**u**];** //次大值就为原先最大值  first**[**u**]=**first**[**v**]+**edge**[**i**].**len**;**  **}**  **else** //试试能不能更新次大值  second**[**u**]=**max**(**second**[**u**],**first**[**v**]+**edge**[**i**].**len**);**  **}**  **}**  ans**=**max**(**ans**,**first**[**u**]+**second**[**u**]);**  **}**  int main**()**  **{**  int u**,**v**,**len**;**  **while(~**scanf**(**"%d%d%d"**,&**u**,&**v**,&**len**))**  **{**  AddEdge**(**u**,**v**,**len**);**  AddEdge**(**v**,**u**,**len**);**  **}**  Dp**(**1**,**0**);**  cout**<<**ans**<<**endl**;**  **return** 0**;**  **}** |

# 魔板问题

【算法分析】

作为算法设计，我们将问题的要求改选为300步的操作序列之内的穷举，将操作序列的全部列举构成一个树状结构（用X表示初始状态）如图10.27所示。

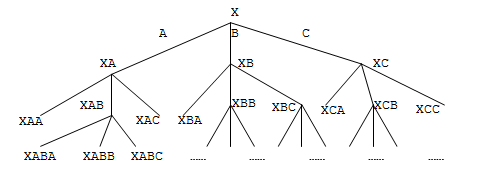


图10.27

如果按题目给定的300步的范围计算，实际可能产生的结点数目将是十分惊人的，所以要将一些不合题意的结点剪枝，经过进一步分析，会发现有些操作序列是循环的，例如XAA＝X，XBBBB＝X，XCCCC＝X，…，所以我们需要记录下不同的结点序列，一旦遇到相同的序列则删除。

例如可以用如图10.28所示的结构来记录每一步操作。

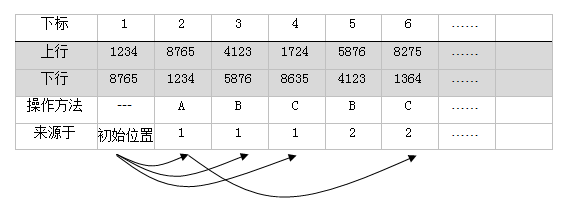


图10.28

这道题跟八数码问题类似，也就是用康托展开式压缩状态，保证每种状态不会重复出现，然后进行宽搜即可。但是由于它说需要输出字典序最小的操作序列，所以最好用宽搜而不是双向广搜。

参考代码如下所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138 | //魔板问题 — 宽搜  #include <bits/stdc++.h>  using namespace std;  bool have[45000]= {0}; //存储某状态是否存在  char did[45000]= {0}; //存储某一步的操作  int bfs[45000]= {0},last[45000]= {0},cost[45000]= {0};  //分别存储当前状态，上一个状态的序号，总共进行了多少次操作  int s[8]= {1,2,3,4,8,7,6,5},ans=0,enter=0,now=0,len=0,f=0,temp=0;  /\* s数组表示魔板的状态，ans表示目标状态，enter是为了保证输出  每60个一行，now是目前搜索到的序号，len是目前总共状态，f表示  答案的序号 \*/  int cantor() //用康托展开式压缩状态  {  int a=0,i=0,j=0,b=0;  for(i=7; i>=1; --i)  {  b=s[i]-1;  for(j=7; j>i; --j)  if(s[j]<s[i])  --b;  a+=b;  a\*=i;  }  return a;  }  int change() //把状态压缩成8位的数字  {  int a=0,i=0;  for(i=0; i<=7; ++i)  a=a\*10+s[i];  return a;  }  void turn(int num) //把数字解压  {  int i=0;  for(i=7; i>=0; --i)  {  s[i]=num%10;  num/=10;  }  return;  }  void go(char tmp)  {  ++len;  bfs[len]=change();  temp=cantor();  if(have[temp])  --len;  else  {  have[temp]=1;  cost[len]=cost[now]+1;  last[len]=now;  did[len]=tmp;  if(bfs[len]==ans)  f=len;  }  }  void go1() //操作A  {  int i=0;  for(i=0; i<4; ++i) //进行操作  {  temp=s[i],s[i]=s[i+4],s[i+4]=temp;  }  go('A'); //尝试添加  for(i=0; i<4; ++i) //还原  {  temp=s[i],s[i]=s[i+4],s[i+4]=temp;  }  }  void go2() //操作B  {  temp=s[3],s[3]=s[2],s[2]=s[1],s[1]=s[0],s[0]=temp;//进行操作  temp=s[7],s[7]=s[6],s[6]=s[5],s[5]=s[4],s[4]=temp;//尝试添加  go('B');  temp=s[0],s[0]=s[1],s[1]=s[2],s[2]=s[3],s[3]=temp;//还原  temp=s[4],s[4]=s[5],s[5]=s[6],s[6]=s[7],s[7]=temp;  }  void go3() //操作c  {  temp=s[1],s[1]=s[5],s[5]=s[6],s[6]=s[2],s[2]=temp;//进行操作  go('C'); //尝试添加  temp=s[2],s[2]=s[6],s[6]=s[5],s[5]=s[1],s[1]=temp;//还原  }    void out(int num) //输出  {  if(num!=1)  {  out(last[num]);  cout<<did[num];  ++enter;  if(enter%60==0)  cout<<"\n"; //保证每行最多60个字符  }  }  int main()  {  int i=0,j=0;  have[cantor()]=1; //确定初始状态  bfs[1]=12348765;  now=1;  len=1;  cin>>s[0]>>s[1]>>s[2]>>s[3]>>s[7]>>s[6]>>s[5]>>s[4];//保持输入顺序的正确  ans=change();  if(ans==bfs[1])  f=1;  while(f==0 && now<=len) //宽度优先搜索  {  turn(bfs[now]);  go1();  if(f==0)  go2();  if(f==0)go3();  ++now;  }  if(f==0)  cout<<-1<<"\n";  else  {  cout<<cost[f]<<"\n";  out(f);  if(enter%60!=0)  cout<<"\n"; //保证最后一行有回车  }  return 0;  } |

更精简代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69 | //魔板问题-STL  #include<bits/stdc++.h>  using namespace std;  string goal;  map<string,string>m;  queue<string>q;  void A(string x)  {  string t=x;  for(int i=0; i<4; i++)  swap(x[i],x[7-i]);  if(!m.count(x)) //没有出现过  {  q.push(x);  m[x]=m[t]+'A';  }  }  void B(string x)  {  string t=x;  x[0]=t[3],x[1]=t[0],x[2]=t[1],x[3]=t[2],x[4]=t[5],x[5]=t[6],x[6]=t[7],x[7]=t[4];  if(!m.count(x))  {  q.push(x);  m[x]=m[t]+'B';  }  }  void C(string x)  {  string t=x;  x[1]=t[6],x[2]=t[1],x[5]=t[2],x[6]=t[5];  if(!m.count(x))  {  q.push(x);  m[x]=m[t]+'C';  }  }  int main()  {  freopen("Magic.in","r",stdin);  freopen("Magic.out","w",stdout);  char c;  for(int i=0; i<8; i++)  {  cin>>c;  goal+=c;  getchar();  }  q.push("12345678");  m["12345678"]="";  while(!q.empty())  {  A(q.front());  B(q.front());  C(q.front());  if(m.count(goal)!=0)  break;  q.pop();  }  cout<<m[goal].size()<<endl<<m[goal]<<endl;  return 0;  } |