第1章 深度优先搜索的优化

深度优先搜索是一种应用很广的搜索算法，十分适合求可行解、最优解之类的问题，并且拥有远低于广度优先搜索的空间复杂度。但作为代价，其时间复杂度一般为指数级别，因此直接的、不加优化的深度优先搜索时间效率往往很低，更是难以应付信息学竞赛严格的运行时间限制。因此，我们经常在运用它时加上各种优化。

深度优先搜索是一种应用很广的搜索算法，但其复杂度一般为指数级别。

本章我们讨论的内容，就是在建立深度优先搜索算法的结构之后，对它加上各种时间上的优化。

1.1 剪枝优化

【知识讲解】

我们先想一想 NOIP2001 提高组第二题（数的划分）：将整数 $n(6<n\leqslant 200)$ 分成 $k(2\leqslant k\leqslant 6)$ 份，且每份不能为空，任意两个方案不相同（不考虑顺序）。求有多少种不同的分法。例如，$n=7$，$k=3$ 时有 4 种分法。

这个问题就是要我们求一个长度为 k 单调不减的数列 $\{a_k\}$ 的数目，使得 $a_1+a_2+\cdots+a_k=n$。显然，我们可以暴力深度优先搜索：设 dfs(n,k,x) 表示将 n 划分成 k 份，且 $a_k\leqslant x$ 的方案数。那么我们就可以枚举 a_k 的值，然后去到 $k-1$ 的层；而当 $k=0$ 时，dfs(n,k,x)=[n=0]。

但这样的复杂度很高。注意到我们可能重复地算参数相同的 dfs(n,k,x) 的值很多次，而恰好 n、k、x 的值都不会很大，所以我们可以借鉴动态规划的思想，记录已经算过的值，比如设一个三维数组 f，令 $f[n][k][x]$=dfs(n,k,x)；而我们进入 dfs(n,k,x) 时，如果 $f[n][k][x]$ 有值，则直接返回它的值。这就是记忆化剪枝。

直接记忆化，理论复杂度上界是 $O(nk^3)$（状态数为 $O(nk^2)$，每个状态的转移数为 $O(k)$），可以过掉这题了；但是我们也可以想想继续优化。我们可以先把合法数列的定义转换一下，改为单调不增，这也是等价的。这样一来，我们设状态时就可以省掉 x 这一维，直接设 dfs(n,k) 表示将 n 划分成 k 份的方案数，并且只有两种转移：第一种是让 $a_1\sim a_k$ 整体 +1，k 不变；第二种是让 a_k+1，$k-1$。此时，我们再加上记忆化剪枝，复杂度上界就仅为 $O(nk)$ 了。

我们可以审视一下深度优先搜索的本质：我们把状态（或者用博弈论里的术语，局面）抽象成一个点，把一个转移（或者说操作）抽象成一条有向边，那么如图 1-1 所示，它就是从一个（或多个）根出发，沿着可以转移的边生成一个 DAG（多数时候，或者使用记忆化剪枝时，它是一棵树，我们可以称之为搜索树），并且要以某种顺序遍历它，找到某些合法的目标结点并计算答案。

而剪枝，就是通过某些判断，避免一些不必要的遍历过程；形象地说，就是剪去搜索树中的某些枝条。

显然，应用剪枝优化的核心问题是设计剪枝判断方法，并且首要条件是剪枝必须正确。在上面的题中，我们运用了深度优先搜索的一种十分常用的剪枝（剩下一种是状态和转移的优化，并非剪枝）；而深度优先搜索的剪枝大致可以归为以下几类。

(1) 记忆化剪枝：记录参数相同的返回值，保证每个状态只会被计算一次。有时也可以只记录上一次的操作之类的信息，来尽量避免对同一个状态重复计算，达到类似记忆化剪枝的效果。

(2) 最优性剪枝：对于一类要求让某个值极大 / 极小的问题，若我们可以在搜索的过程中以不高的复杂度顺便算出这个值，那我们可以记录答案；当我们走到已经不比当前答案更优的点时，我们就可以不从它继续往下遍历。

(3) 可行性剪枝（上下界剪枝）：这个适用于一类有条件限制的搜索问题，在搜索过程中不断

地判断能否从当前点遍历到一个合法的目标结点，如不行则不遍历。一般来说，都是将当前的花费＋还需至少的花费与要求花费比较，或者将当前的花费＋还需至多的花费与要求花费比较。这种剪枝借鉴了 A* 的估价函数的思想。

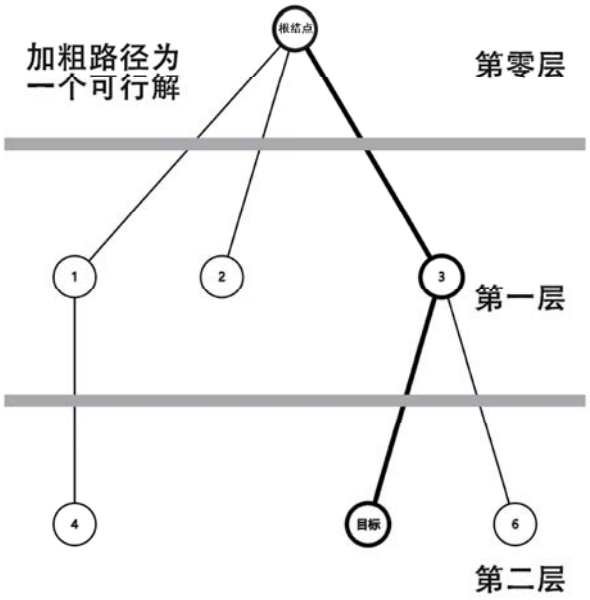

图 1-1　搜索算法的图示 (1)

(4) 搜索顺序剪枝：这个适用于只需搜索到一个合法目标结点的问题，或者可以和最优性剪枝配合发挥良好效果的问题，剪枝方法即调整遍历顺序。但这是一种比较玄妙的剪枝，如使用不当可能反而会拖慢时间复杂度。

例 1　吃奶酪 (cheese，1s，256MB)

【问题描述】

房间里放着 n 块奶酪。一只小老鼠要把它们都吃掉，问至少要跑多少距离？老鼠一开始在 (0,0) 点处。

【输入格式】

第 1 行一个数 n ($n \leqslant 15$)。

接下来每行两个实数，表示第 i 块奶酪的坐标。($0 \leqslant \text{abs}(x),\text{abs}(y) \leqslant 10\,000$)

【输出格式】

一个数，表示要跑的最少距离，保留两位小数。

【输入样例】

```
4
1 1
1 -1
-1 1
-1 -1
```

【输出样例】

```
7.41
```

【问题分析】

首先，我们肯定是每一次直奔一块奶酪去；也就是说，我们经过的点只有原点和有奶酪的点。那么，只要会两点距离公式 $\mathrm{dis}(p,q)=\sqrt{(x_p-x_q)^2+(y_p-y_q)^2}$，我们就可以列出一个暴力的深度优先搜索了：开一个桶标记吃掉了的奶酪，每次枚举一个还未吃掉的奶酪，走过去吃完所有 n 个奶酪就更新答案。

但是这样的复杂度是 $O(n!)$ 的。我们可以加一个最优性剪枝：我们时刻记着可能的答案（不比真正的答案更优，但可能是真正的答案），若走到某个状态时，花费已经大于那个可能的答案了，那便直接退出，不继续往下搜索。

这样的剪枝有点玄妙，理论复杂度并没有优化，但实际效率却可以高不少。

【参考程序】

```
#include<cmath>
#include<cstdio>
#define sqr(x) (x)*(x)
#define fo(i,a,b) for(int i=a;i<=b;i++)
#define dis(p,q) sqrt(sqr(x[p]-x[q])+sqr(y[p]-y[q]))
int n;
double x[16],y[16],ans;
bool vis[16];//vis[i] 表示 i 号奶酪有没有被吃
void dfs(int now,double sum)//now 表示当前所在的点，sum 表示当前的花费
{
  if(sum>ans) return;// 最优性剪枝
  bool ed=true;//ed 表示是否已将所有奶酪吃完
  fo(i,1,n) if(vis[i]==false)
  {
        ed=false;
        vis[i]=true;
        dfs(i,sum+dis(now,i));
        vis[i]=false;
  }
  if(ed) ans=sum;
}
int main()
{
  freopen("cheese.in","r",stdin);
  freopen("cheese.out","w",stdout);
  scanf("%d",&n);
  fo(i,1,n) scanf("%lf%lf",&x[i],&y[i]), ans+=dis(i-1,i);// 先算出一个可能的答案
  dfs(0,0);// 可以把原点视为 0 号奶酪
  printf("%.2lf",ans);
}
```

【实践巩固】

例2　数字三角形 (triangle，ls，256MB)

【问题描述】

有这么一个游戏：

写出一个1至$N(1 \leqslant N \leqslant 10)$的排列$a_i$，然后每次将相邻两个数相加，构成新的序列，再对新序列进行这样的操作，显然每次构成的序列都比上一次的序列长度少1，直到只剩下一个数字。

现在已知最后的那个数字是sum，求一个合法的最初序列。若答案有多种可能，则输出字典序最小的那一个。若无解，则不输出。

【输入格式】

一行两个整数：N、sum。

【输出格式】

一行N个整数，字典序最小的合法最初序列。无解则不输出。

【输入样例】

```
4 16
```

【输出样例】

```
3 1 2 4
```

1.2　迭代加深优化

【知识讲解】

先考虑一个问题模型：一个初始状态，多个合法的结束状态，每个状态都有4种转移，求初始状态转移到任意一个结束状态最少需要多少步。

考虑搜索。我们可以把搜索树大致地画成如图1-2所示的模样。

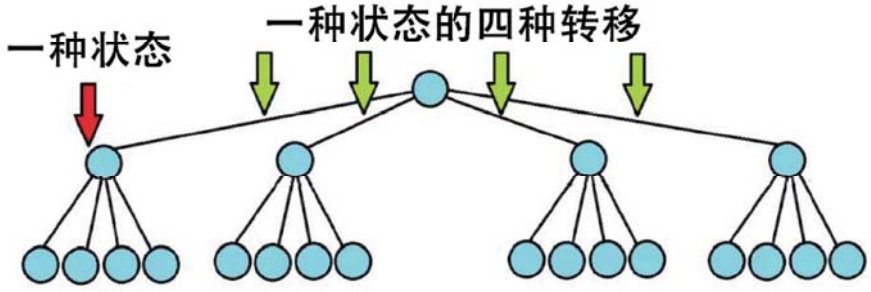

图1-2　搜索算法的图示(2)

如果我们直接采用深度优先搜索的话，很有可能会陷于不妙的境地：如图1-3所示，我们可

能在不能去到合法的目标状态的子树里搜了很久，因为它们的深度很可能非常大（在某些题中，这个结点深度甚至是无限的——因为它可以一直在一个由转移边组成的环里不停地转圈），而这个搜索耗费大量时间却徒劳无功；而一个合法的结束状态的深度其实并不深，只是恰巧不在先搜到的子树内。

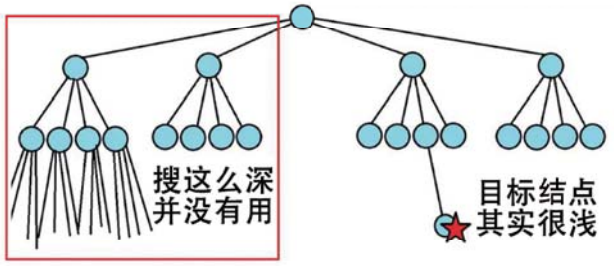

图 1-3　深度优先搜索的缺点

当然我们也可以考虑广度优先搜索（见图 1-4）。我们开一个队列，从左到右枚举队首，将它的四种转移统统加到队尾，这样一层一层地推进，实际上也就是一步一步地增加，因而首次到达的结束状态的深度就是最少的步数。

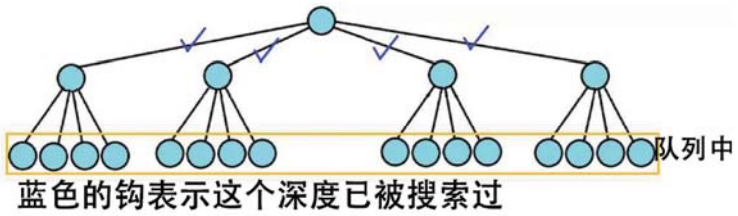

图 1-4　广度优先搜索的图示

但广度优先搜索也有一个问题：如果最浅的结束状态稍微深一点，用于保存状态的队列所需的空间就会指数级增长。以本题为例，如果到达结束状态最少要 10 步，队列需要存储的状态数就有 $1+4+4^2+4^3+\cdots+4^{10}=1\ 398\ 101$ 个。并且在某些题目中，存储一个状态的空间并非 $O(1)$，而是一个很大的数（比如要存一个长为 n、宽为 m 的矩阵，甚至要开 $O(nm)$ 的空间），那就更加崩溃了。

相形之下，深度优先搜索就没有这个难题，因为一般而言我们采取深度优先搜索时只需记录当前状态的值，最不济也只是记录初始状态到当前状态一条链上的值。若单个状态的空间复杂度是 $O(k)$，那么深度优先搜索的空间复杂度便仅有 $O(k)$ 或 $O(\text{deep}\times k)$（deep 为最浅的结束状态的深度）。

深度优先搜索、广度优先搜索各有优劣。迭代加深 (iterative deepening，ID) 则结合两者的优点，用深度优先搜索模拟广度优先搜索。它的主要思想就是每次深度优先搜索之前限定一个深度限制，要求深度优先搜索不得超过这个深度。每一次深度优先搜索，我们检查当前所处的深度。如果达到深度限制，立刻返回。一次完整的深度优先搜索完成，如果没有找到答案，增加限定的深度，继续 ID。ID 的流程大致如图 1-5 所示。

运用 ID，无论是深度优先搜索的搜索过深问题还是广度优先搜索的空间过大问题都迎刃

而解。

现在唯一的问题就是，我们没有用空间保存上一层的信息，每次 ID 都要把上一层重新搜索一遍，这是否会大大增加时间开销？

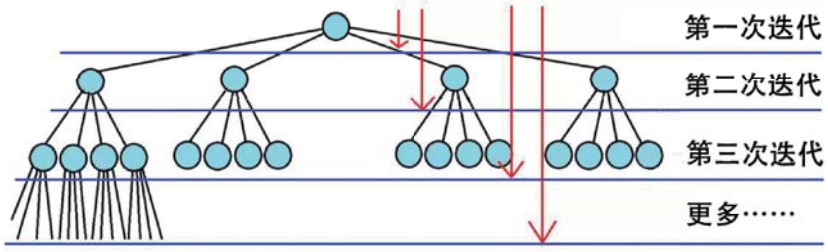

图 1-5　迭代加深的图示

实际上并无此问题。我们可以设 $D(x)$ 表示深度限制为 x 时搜索树内的结点数，设 k 表示每个结点的出边数（即单一状态的转移数，不妨假设所有状态的转移数都相等）。则 $D(x)=1+k+k^2+k^3+\cdots+k^x=\frac{k^{x+1}-1}{k-1}$。如果答案的深度为 h，那我们总共搜索过的结点总数即为 $D(0)+D(1)+D(2)+\cdots+D(h)=\frac{\frac{k(k^{h+1}-1)}{k-1}-(h+1)}{k-1}=\frac{k}{k-1}\cdot\frac{k^{h+1}-1}{k-1}-\frac{h+1}{k-1}\approx\frac{k}{k-1}D(h)$。可以发现，最坏情况下 $\frac{k}{k-1}=2$（此时 $k=2$，$k=1$ 时不需要搜索），完全可以视为一个常数。这样一来，我们就证明了，ID 是结合了深度优先搜索的空间复杂度和广度优先搜索的时间复杂度的奇妙算法。

当然，我们在采用 ID 时，也可以套上普通深度优先搜索的各种剪枝。一般来说，如果 ID 再套上一个可行性剪枝（即时刻判断当前深度 + 估价函数值（即还需至少的步数）是否大于深度限制，如果大于则退出），我们就把这个称为 IDA*。

例 3　The Rotation Game(rotation，2s，64MB)

【问题描述】

现有一块有 24 个格子的井字板子，每个格子用 1、2 或 3 标记，每种格子各有 8 个。起初这些格子分布随机，你需要通过 A ~ H 8 种操作将中心 8 个格子变为相同的标记。(图 1-6 中使用 A 操作将 A 列向上拉了一格，使用 C 操作将 C 列向右拉了一列，中心变为 2)

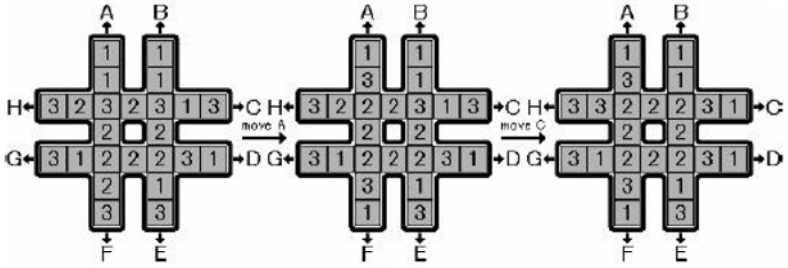

图 1-6　The　Rotation　Game 题目问题描述

【输入格式】

有多组数据 (≤ 30), 每组数据包含一行 24 个数字, 代表从左上到右下 24 个格子的初始状态。输入 0 代表结束。

【输出格式】

每组数据包含两行，第 1 行是最佳的操作顺序，第 2 行是此时中心的字符。若不需要操作，即初始时中心 8 个字符就相同，则输出 No moves needed(也要输出中心字符)。

最佳操作顺序为：操作次数最少。同次数若有多种则为字典序小者。

【输入样例】

```
1 1 1 1 3 2 3 2 3 1 3 2 2 3 1 2 2 2 3 1 2 1 3 3
1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3
0
```

【输出样例】

```
AC
2
DDHH
2
```

【问题分析】

首先，我们可以用常量数组将 8 种操作涉及的各 7 个格子以及中心的 8 个格子记录下来，并记录每种操作的逆操作，然后便可以开始 ID：每次按 A ~ H 的顺序枚举一种操作，直接移动，回溯时则按它的逆操作移动。每一次检查中心格是否都相等即可。

可以加两个比较明显的剪枝：第一个是记录上一次的操作，不进行上一次操作的逆操作 (减少原地打转的次数，类似于记忆化剪枝)；第二个是设计一个估价函数，当前状态的估价可以 =8- 中心格中众数的出现次数 (因为这是至少还需进行的操作数)，如果当前的深度 + 估价 > 深度限制，则可以退出。第二个也就是可行性剪枝。

【参考程序】

```
#include<cstdio>
#include<algorithm>
#define fo(i,a,b) for(int i=a;i<=b;i++)
using namespace std;
const int cen[8]={6,7,8,11,12,15,16,17};            //8 个中心点
const int inv[8]={5,4,7,6,1,0,3,2};                 // 每种操作的逆操作
const int o[8][7]={{0,2,6,11,15,20,22},             //AF
                   {1,3,8,12,17,21,23},             //BE
                   {10,9,8,7,6,5,4},                //CH
                   {19,18,17,16,15,14,13},          //DG
                   {23,21,17,12,8,3,1},             //EB
                   {22,20,15,11,6,2,0},             //FA
                   {13,14,15,16,17,18,19},          //GD
                   {4,5,6,7,8,9,10}};               //HC
int a[24],lim,c[4],mc;
char ans[110];
void calc()// 计算中心格中众数的出现次数
{
```

```
  c[1]=c[2]=c[3]=0;
  fo(i,0,7) c[a[cen[i]]]++;
  mc=max(c[1],max(c[2],c[3]));
}
void move(int k)//移动操作
{
  int tmp=a[o[k][0]];
  fo(i,0,5) a[o[k][i]]=a[o[k][i+1]];
  a[o[k][6]]=tmp;
}
bool dfs(int t,int la)
{
  calc();
  if(t+8-mc>lim) return 0;//可行性剪枝
  if(mc==8)
  {
        ans[t]=0;
        printf("%s\n%d\n",ans,a[6]);
        return 1;
  }
  fo(i,0,7) if(inv[i]^la)
  {
        ans[t]=i+65;
        move(i);
        if(dfs(t+1,i)) return 1;
        move(inv[i]);
  }
  return 0;
}
int main()
{
  freopen("rotation.in","r",stdin);
  freopen("rotation.out","w",stdout);
  for(scanf("%d",a);a[0];scanf("%d",a))
  {
        fo(i,1,23) scanf("%d",&a[i]);
        calc();
        if(mc==8)
        {
              puts("No moves needed");
              printf("%d\n",a[6]);
              continue;
        }
        for(lim=1;!dfs(0,-1);lim++);
  }
}
```

【实践巩固】

例4　水叮当的舞步 (dance，1s，256MB)

【问题描述】

水叮当得到了一块五颜六色的格子形地毯作为生日礼物；更加特别的是，地毯上格子的颜

色还能随着踩踏而改变。

为了讨好她的偶像虹猫，水叮当决定在地毯上跳一支轻盈的舞来卖萌。

地毯上的格子有 N 行 N 列，每个格子用一个 0 ~ 5 之间的数字代表它的颜色。

水叮当可以随意选择一个 0 ~ 5 之间的颜色，然后轻轻地跳动一步，地毯左上角的格子所在的连通块里的所有格子就会变成她选择的那种颜色。这里连通定义为：两个格子有公共边，并且颜色相同。

由于水叮当是施展轻功来跳舞的，为了不消耗过多的真气，她想知道最少要多少步才能把所有格子的颜色变成一样的。

【输入格式】

每个测试点包含多组数据。

每组数据的第 1 行是一个整数 N，表示地毯上的格子有 N 行 N 列。

接下来一个 $N \times N$ 的矩阵，矩阵中的每个数都在 0 ~ 5 之间，描述了每个格子的颜色。

N=0 代表输入的结束。

【输出格式】

对于每组数据，输出一个整数，表示最少步数。

【输入样例】

```
2
0 0
0 0
3
0 1 2
1 1 2
2 2 1
0
```

【输出样例】

```
0
3
```