搜索优化

1.1 迭代加深搜索

深度搜索的一种，即限定深搜的深度，对于一些可以用深搜解决，解答树的深度没有明显上限，可以考虑迭代加深搜索。具体在搜索过程中，限定搜索层数，当达到规定层数时回溯，不停增加限定层数以达到目标为止。

1.1.1 例 1：埃及分数 P1022

分析：本题由于搜索层数不明，用深搜极易陷入死胡同，用广搜空间又吃不消，这时迭代加深搜索就成了考虑的对象。确定了搜索模式之后，我们容易得到以下两个基本思路：

1. 枚举对象：分母

$\frac{a}{b}=\frac{1}{a_1}+\frac{1}{a_2}+...+\frac{1}{a_n}$, 设 $a_1<a_2<...<a_n$

2. 剪枝手段

设限定的搜索层数为 D，当前搜到第 C 层，当前正要枚举分母 a_k，还需枚举总和为 x/y 的分数。ans[D] 表示当前最优解中的第 D 个分母，如果还没有得到解则表示正无穷。则必然有：

$$max(\lfloor\frac{y}{x}\rfloor,a_{k-1})+1\le a_k\le min(\lfloor(D-C+1)*\frac{y}{x}\rfloor,\lfloor\frac{maxin}{x}\rfloor,ans[D]-1)$$

枚举的初值容易得出，但终值的确定则要用到我们一开始对分母有序性的假设了。值得注意的是，直接限界避免了搜索过程中屡次使用可行性剪枝，在一定程度上可以提高了程序的运行速度。

参考代码：

```
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;
typedef long long LL;
const int maxn = 1000;
LL ans[maxn],v[maxn];
```

```
int maxd,kcase;
int gcd(LL a,LL b){// 最大公约数，用于约分
    LL temp;
    while(b){temp = b;b = a % b;a = temp;}
    return a;
}
LL get_first(LL a, LL b){//满足1/c<=a/b的最小c
    return b/a + 1;
}
bool better(int d){//更新最优值：如果当前解v比目前最优解ans更优，更新ans
    for(int i=d;i>=0;i--)
        if(v[i] != ans[i])
            return ans[i] == -1 || v[i] < ans[i];
    return false;
}
//当前深度为d，分母不能小于from，分数之和恰好为aa/bb
bool dfs(int d,int from,LL aa,LL bb){
    if(d == maxd){//深度达到当前枚举个数
        if(bb % aa) return false;//aa/bb必须是埃及分数
        v[d] = bb/aa;//保存分母
        if(better(d)) memcpy(ans,v,sizeof(LL)*(d+1));//更新最优解
        return true;//返回成功
    }
    bool ok = false;//用于返回本次递归的结果
    from = max((LL)from,get_first(aa,bb));//枚举的起点
    for(int i=from; ;i++){
        if(bb * (maxd+1-d) <= i*aa) break;//剪枝：如果剩下的maxd+1-d个分数全部都是1/i，加起来仍然不超过aa/bb，则无解！ (maxd+1-d)/i <= aa/bb
        v[d] = i;//保存分母
        //计算aa/bb - 1/i，设结果为a2/b2
        LL b2 = bb*i;
        LL a2 = aa*i - bb;
        LL g = gcd(a2,b2);//用于约分
        if(dfs(d+1,i+1,a2/g,b2/g)) ok = true;//找到解返回成功
    }
return ok;}

void solve(int a,int b){
    int ok = 0;
    for(maxd = 1; ;maxd++){//枚举深度，即等式相加的个数
        memset(ans,-1,sizeof(ans));
        if(dfs(0,get_first(a,b),a,b)){//递归找到时返回成功
            ok = 1;break;//标记并退出枚举
```

```
        }
    }
  if(ok) {
    for(int i=0;i<maxd;i++)
    printf("%lld ",ans[i]);
    printf("%lld\n",ans[maxd]);
  }
  else printf("%d/%d\n",a,b);
}
int main(){
    int a,b;
    while(scanf("%d%d",&a,&b)!=EOF)
        solve(a,b);
    return 0;
}
```

code/p1022.cpp

分块算法求解问题主要从**以下几个方面去思考：**

1、不完整块的处理

2、中间完整块的处理

3、预处理数据（复杂度一般在 $n\sqrt{n}$ 内）

1.2 启发式搜索

启发式搜索的主要目标是使用一个函数去判断所有状态的“好坏”，以提高搜索找到解的效率。

通常，估价函数表示成一个函数或是一个状态，这个函数叫做“估价函数”。对于相同的题目，有时有不同的估价函数。直观地看，越优秀的估价函数，搜索的速度就越快。当然，估价函数不一定是十全十美的（否则就是贪心法了），总归会对某些状态予以不太准确的评价，于是，评价值（函数返回值）和实际好坏的差异越小，估价函数就越优秀。

（1）启发式剪枝

最简单也是最常用的启发式搜索是利用估价函数来剪枝。假设我们的问题是要求找最小总花费。对于一个可接受的估价函数，当前花费是 A，启发函数返回了 B，当前子问的最优解是 A+B。如果找到的一个解一个花费是 C，C<A+B，这个状态就不必要

搜索了。

这样编写和调试也比较简单（假设一个状态需要长时间而被剪掉……），且可以极大地提高程序效率。它对 DFS 尤其有效。

（2）最佳优先搜索

这种方法好比就是贪心算法。

每次不扩展所有子结点，而是按“好坏程度”来扩展。与贪心不同的是，贪心只尝试“最优”路径，但是 BFS 首先扩展“希望大”的，再扩展“希望小”的，如果结合上述描述，搜索会得到很好的结果。

（3）A* 法

A* 法是类似贪心的 BFS。

BFS 一般扩展最小耗费的点。A* 算法在另一方面，扩展最有希望的点（估价函数返回值最优）。状态被保存在一个优先队列中，按照 Cost* 价值排列。每一次，程序处理最低优先的点，且把它的孩子按照适当顺序处理。对于一个可容许的估价函数，第一个找到的状态保证是最优的。

启发函数的表示：f’(n)=g’(n) + h’(n) 表示估计的该结点的好坏。

一般越小代表这个点越优。其中 g’(n) 表示从初始结点到 n 的路径长度,h’(n) 表示从 n 到目标结点的距离的估计。

A* 算法每次选择 f’ (n) 最小的点进行拓展 (dijkstra 相当于 h’ (n)=0)

当 h’(n)=h(n) 时 (h(n) 表示从 n 到目标点实际最短距离)，A* 算法变成了贪心算法

当 h’(n)=0 时，A* 算法成了 dijkstra 算法

若搜索图中边权非负，且满足 h’(n)<=h(n)，则 A* 保证第一次搜索到目标结点便确定了最优解。但搜索数量会较大。

若存在 h’(n)>h(n), 则搜索数量会减少，但不一定能找到最优解。所以，在保证 h’(n)<=h(n) 的情况下越大越好。比如对于每次能向四个方向走一步的棋盘，h’(n) 可以选择两点间欧几里得距离。

简而言之：设计一个估价函数 h’ (n), 让它小于等于 n 到终点的实际距离 h(n)。

证明：启发式搜索一定是最优解么？

证：当 H（n）小于等于真实距离是真实解，因为通过 F（n）的比较每次被临时放弃的节点都是因为自己的 F（n）比别人的大，而自己的 G（n）是真实距离，H（n）又小于等于真实距离，所以这个被放弃的点在这一步的确不是最优解，直到它可能是最优解的时候它才会被搜索。故得证。

1.2.1 例 2：K 短路

一个 n 个节点，m 条边的有向图，求 S-T 的第 k 短路的长度，路径允许重复经过点或边。

【分析】

（1）直接求解：

用 bfs+ 优先队列，求 k 次到 T 的距离，当 T 第 k 次被取出就得到答案，时间复杂度：O(k*(N+M)*log(n+m))

（2）用 A* 算法优化：

根据估价函数的原则：第 k 短路从 x 到 T 的预估距离 f(x) 不大于第 k 短路 x 到 T 的实际距离 g(x)，所以可以把 f(x) 定为 x 到 T 的最短路，这样保证 f(x)<=g(x)，同时也是 g(x) 的变化趋势，于是得到如下算法：

1、预处理每个节点 x 到 T 的最短路 f(x)（反图跑最短路算法）

2、建立优先堆，存入二元组 (x,dist+f(x)),dist 表示起点到 x 走的路，初值（S,0+F(S)）然后扩展 x 的每条边对于的点 y，如果 y 没有被取出 k 次，就把新二元组（y,dist+len(x,y)+f(y)）入堆

3、直到第 k 次取出包含 T 的二元组

参考代码：

```
#include<bits/stdc++.h>
#define maxn 5010
#define maxm 201000
#define inf 1e12

using namespace std;

struct yts{
  int now;
```

```
  double g;
};

double dis[maxn];
int q[maxn];
bool vis[maxn];

bool operator<(yts x,yts y){
  return x.g+dis[x.now]>y.g+dis[y.now];
}

priority_queue<yts> Q;

int head[maxn],to[maxm],next[maxm];
double len[maxm],Len[maxm];
int Head[maxn],To[maxm],Next[maxm];
int n,m,num,ans,x,y,S,T,Num;
double e,z;

void addedge(int x,int y,double z){
  num++;to[num]=y;len[num]=z;next[num]=head[x];head[x]=num;
}

void add(int x,int y,double z){
  Num++;To[Num]=y;Len[Num]=z;Next[Num]=Head[x];Head[x]=Num;
}

void spfa(){
  for (int i=S;i<=T;i++) dis[i]=inf;
  dis[T]=0;vis[T]=1;q[1]=T;
  int l=0,r=1;
  while (l!=r){
    l++;if (l==maxn) l=0;
    int x=q[l];
    for (int p=Head[x];p;p=Next[p])
      if (dis[x]+Len[p]<dis[To[p]]){
        dis[To[p]]=dis[x]+Len[p];
        if (!vis[To[p]]) {
        r++;if (r==maxn) r=0;
          vis[To[p]]=1;q[r]=To[p];
        }
      }
    vis[x]=0;
  }
}
```

```
55
56 void Astar(){
57   Q.push((yts){S,0});
58   while (!Q.empty()){
59     yts x=Q.top();Q.pop();
60     if (x.now==T){
61       e-=x.g;
62       if (e<0) return;
63       ans++;
64       continue;
65     }
66     if (x.g+dis[x.now]>e) continue;
67     for (int p=head[x.now];p;p=next[p])
68  Q.push((yts){to[p],x.g+len[p]});
69   }
70 }
71 int main(){
72   scanf("%d%d%lf",&n,&m,&e);
73   for (int i=1;i<=m;i++){
74     scanf("%d%d%lf",&x,&y,&z);
75     addedge(x,y,z);add(y,x,z);
76   }
77   S=1,T=n;
78   spfa();
79   Astar();
80   printf("%d\n",ans);
81   return 0;
82 }
```

code/ksort.cpp

1.2.2 例 3：八数码 P1009

本题很适合练习各类搜索算法。

如果采取 A*，搜索移动步数最少，观察发现，每次移动只能把一个数字与空格交换，这样最多把一个数字向它对应的目标位置靠近 1 步。加入每一步移动都是有效的，这样每个状态到目标状态的移动步数不可能小于所有数字当前位置与目标位置的曼哈顿距离之和。于是我们可以得到一个这样的估价函数：

F(s)=sum(|xi-xm|+|yi-ym|),xi,yi 表示每个数字现在的位置，xm,ym 表示每个数字目标位置。

类似的，很多迷宫类问题的估计函数也是曼哈顿距离。

参考代码：

```
#include<bits/stdc++.h>
using namespace std;
typedef int State[9];
#define ML 100000
struct Node{
  State s;//节点状态
  int fa,p,dep,g;//fa父亲，dep深度，p：0的位置
  int operator<(const Node&t)const{
    return g>t.g;
  }
}List[ML+1];
priority_queue<Node>q;
State goal;//状态数组
int last=0,ans;
int dx[4]={0,-1,0,1};
int dy[4]={-1,0,1,0};
int vis[362880]={0},fact[9]={0};
//函数判断，相当于一个无重复键值的hash函数
int fPD(const State&s){//康拓展开
  int code = 0;//把每个状态映射到code
for(int i=0;i<9;i++){
        int cnt = 0;
        for(int j=i+1;j<9;j++)//看这9个数有多少个逆序对
          if(s[j] < s[i]) cnt++;
        code +=fact[8-i]*cnt;
  }
  if(vis[code]) return 1;//有冲突
      vis[code] =1;
      return 0;
}
void pt(const State& st){//打印一个八数码
  for(int j=0;j<9;j++){
      printf("%3d",st[j]);
      if((j+1)%3==0) cout<<endl;
    }
    cout<<endl;
}
void write(int k){
      if(k==0)return;
      write(List[k].fa);
  pt(List[k].s);ans++;
```

```
}
int g(const State &s){
  int ret=0;
  int c[9],t[9];
  for(int i=0;i<9;i++){
    c[s[i]]=i;t[goal[i]]=i;
  }
  for(int i=1;i<9;i++)
    ret+=abs(c[i]/3-t[i]/3)+abs(c[i]%3-t[i]%3);
     return ret;
}
void expand(Node &st){
  int x = st.p / 3;
  int y = st.p %3;//取出0的位置行列编号0~2
  for(int i=0;i<4;i++){
    int newx = x+dx[i];//
    int newy = y+dy[i];
    int newp = newx*3+newy;//新0的坐标与位置
    if(newx<3&&newx>=0&&newy<3&&newy>=0){
      Node t=st;
    //  memcpy(t.s,st.s,sizeof(st.s));//复制状态
      t.s[newp] = st.s[st.p];//0的位置变化
      t.s[st.p] = st.s[newp];
      if(fPD(t.s)) continue;

      t.fa = last;//父亲
      t.p = newp;
      t.dep++;
      t.g=t.dep+g(t.s);//已走的步+估价
      q.push(t);
    }
  }
}
int bfs(){
  while(!q.empty()){
    Node t = q.top();q.pop();
    List[++last]=t;
    if(memcmp(goal,t.s,sizeof(goal)) == 0)
    //比较数组，内存比较，cstring中
      return 1;
    expand(t); //扩展当前节点
    if(last>=ML)break;
  }
  return 0;//失败
}
```

```cpp
int main(){
  fact[0]=1;for(int i=1;i<9;i++)fact[i]=fact[i-1]*i;
  Node st;
  for(int i=0;i<9;i++) {
    cin>>st.s[i];
    if(st.s[i]==0) st.p=i;
  }
  for(int i=0;i<9;i++)cin>>goal[i];
  st.fa=0;st.g=g(st.s);st.dep=0;
  fPD(st.s);
  q.push(st);
  if(bfs()){
    write(last);
  }
     else cout<<"NO solution!";
}
```

code/P1009.cpp

1.3 IDA* 搜索

IDA* 搜索 = 估价函数 + 迭代加深搜索。与 A* 一样估价函数的准则：不大于实际步数。将深度限制加强为：当前深度 + 估计步数 > 深度限制即结束搜索。

1.3.1 例 4：flood-it P1021

题意：N*N 的网格，每个格子有一个颜色，每次操作可以将左上角的格子所在连通块染成指定颜色，问最少多少次操作

【分析】

IDA* 算法可以发现如果当前矩阵中除了左上角的连通块之外，共有 M 种颜色，那么还需要的步数不小于 M。如果当前搜索深度 + 估价函数的值 > 深度限制，可以剪枝。

每次寻找左上角的格子所在的连通块耗费的时间常数巨大，可以在这里寻求突破。我们引入一个 N*N 的 v 数组。左上角的格子所在的连通块里的格子标记为 1。左上角连通块周围一圈格子标记为 2，其它格子标记为 0。如果某次选择了颜色 c，我们只需要找出标记为 2 并且颜色为 c 的格子，向四周扩展，并相应地修改 v 标记，就可以不断扩大标记为 1 的区域，最终如果所有格子标记都是 1，就找到了答案。

估价：统计场上还剩下多少颜色，至少要染这么多次才能出解，如果预估次数加上当前次数超过了当前迭代加深的深度限制，剪枝。

如果改变颜色后，左上角格子所在的联通块大小没有改变，可以剪枝，避免来回往复地搜索。

参考代码：

```
#include<iostream>
#include<cstring>
using namespace std;
const int dx[4]={-1,0,0,1},dy[4]={0,-1,1,0};
int a[9][9],v[9][9],f[6],n,ID;

int left(){
  int i,j,temp=0;
  memset(f,0,sizeof(f));
  for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
      if(!f[a[i][j]]&&v[i][j]!=1)
      {f[a[i][j]]=1; temp++;}
  return temp;
}

void dfs(int x,int y,int c){
  v[x][y]=1;
  for(int i=0;i<4;i++){
    int nx=x+dx[i],ny=y+dy[i];
    if(nx<1||ny<1||nx>n||ny>n||v[nx][ny]==1)continue;
    v[nx][ny]=2;
    if(a[nx][ny]==c)dfs(nx,ny,c);
  }
}

int fill(int c){
  int temp=0;
  for(int i=1;i<=n;i++)
    for(int j=1;j<=n;j++)
      if(a[i][j]==c&&v[i][j]==2){
        ++temp;
        dfs(i,j,c);
      }
  return temp;
}
bool IDAstar(int dep){
```

```cpp
  int g=left();
  if(dep+g>ID)return 0;
  if(!g)return 1;
  int rec[9][9];
  for(int i=0;i<=5;i++)
  {
    memcpy(rec,v,sizeof(v));
    if(fill(i)&&IDAstar(dep+1))return 1;
    memcpy(v,rec,sizeof(v));
  }
  return 0;
}

int main(){
  while(cin>>n&&n){
    for(int i=1;i<=n;i++)
      for(int j=1;j<=n;j++)cin>>a[i][j];
    memset(v,0,sizeof(v));
    dfs(1,1,a[1][1]);
    for(ID=0;;ID++)
      if(IDAstar(0))break;
    cout<<ID<<endl;
  }
  return 0;
}
```

code/P1021.cpp

1.4 总结

Dfs 适合暴力求解全解：代码简单，效率不高

Bfs 适合求最短路问题：空间需求大

双向搜索：有效减少搜索量

迭代加深：适合解决限制步数问题，有效节约空间，代码简洁

A* 搜索：优化的 bfs，效率高，空间需求大，

IDA* 搜索：实现难度较低