# 搜索问题浅谈

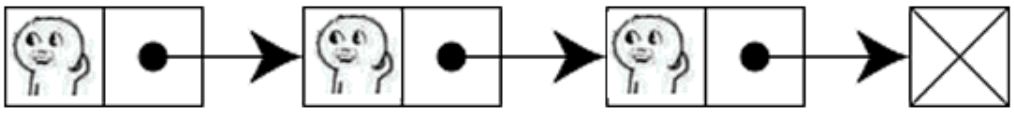
在我看来，搜索是一种最简单也是一种最困难的算法。为什么这样说？首先，作为一种入门算法，搜索所需基本知识很少，而且思考直白简单，任何人都能在短时间之内迅速掌握。然而，由于搜索算法需要枚举所有的情况，所以我们在使用搜索算法时，时间复杂度成为了很大的瓶颈。所以如何选择搜索算法，如何选择搜索顺序，以及如何对一个搜索算法进行剪枝，成为让我们很头疼的问题，更多时候我们不是不会搜，而是不敢搜(大雾)。



首先，我们需要明确什么样的问题看着像搜索？如果题目中某一项的数据范围非常小，在考虑非搜索的”正规算法”时，可以过掉比该数据扩大几倍甚至几十倍的问题时，很可能这题就用搜索来求解。

一般来说，搜索算法大致可以分为盲目搜索和启发式搜索两类，所谓盲目搜索就是我们只能区分当前状态是否为目标状态，而启发式搜索就是指我们需要在搜索过程中加入一

些与问题相关的启发信息，从而指引搜索的方向。这次我主要来跟大家讨论一下与盲目搜索相关的问题，对于启发式搜索的问题，我将会在以后的chat中为大家带来。



# 懵逼链表



## 盲目搜索问题是什么？

盲目搜索最常见的两种姿势就是深度优先搜索(DFS)和广度优先搜索(BFS)。

***DFS***:关于DFS的定义，相信大家在数据结构课本上都不止一次见过，这里就不再赘述了，如果有不太清楚的读者，可以在这里了解一下。那下面我们就通过一个题目来回顾一下DFS。

谈到DFS，我们就不得不提一个很重要的概念，回溯法，简单来说，所谓回溯法就是指选择某一策略向前搜索，当搜索到某一时刻，发现该策略不是最优的或者无法达到我们需要的最终状态，于是我们退回到上一步重新进行选择，下面我们来看几个跟回溯相关的问题。

## 例1：给定一个整数N，$1 \leq N \leq 9$,请打印一下1～N的全排列

什么是全排列呢？举个例子，比如当N=3时，这时N的全排列就有(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)，同时按照字典序输出。

分析:这是我们初学DFS时常遇到的一道题目，非常简单，我们可以定义一个a数组来存每次全排列第i个位置是什么数，然后定义一个vis数组来表示位置i是否被访问过，然后我们进行一下回溯，就可以很容易得出结果。

```
#include "iostream"
#include "cstdio"
#include "cstring"
#include "string"
using namespace std;
const int maxn=100+10;
int n;
int a[maxn];
int vis[maxn];
void dfs(int k){
    if(k==n+1){
        for(int i=1;i<n;i++)
            printf("%d ",a[i]);
        printf("%d\n",a[n]);
    }
    for(int i=1;i<=n;i++){
        if(!vis[i]){
            vis[i]=1;
            a[k]=i;
            dfs(k+1);
            vis[i]=0;
            a[k]=0;
        }
    }
}
int main()
{
    cin>>n;
    dfs(1);
    return 0;
}
```



【变式】搞懂了全排列问题，那我们再来看一个类似的问题，会下国际象棋的人都知道，皇后可以可以攻击同一行，同一列，同一条对角线上的其他棋子，现在我们需要把n $(1 \leq n \leq 10)$个皇后放在棋盘上，使得皇后之间都不互相攻击，请问不同的放置方法有多少种，按字典序输出所有方案，每种方案输出皇后所在的列号。

分析：这就是著名的N皇后问题，当然我们知道这个问题有公式可以直接求解，但是现在我们需要用回溯的方法来做。类比上题，我们应该可以很容易得出想法，既然我们攻击的对象是同一行，同一列，和同一条对角线，于是，我们就可以用4个数组分别来标记一下，是否处于同一行，同一列，同一条对角线。这里补充2个性质，同一条主对角线上的元素横纵坐标之差相同，副对角线上的元素，横纵坐标之和相同，这样我们就可以通过回溯很轻松解决这道题目了。

```
#include "iostream"
#include "cstdio"
#include "cstring"
#include "string"
```

```
using namespace std;
const int maxn=200;
int n;
int a[maxn];
int col[maxn],row[maxn],L[maxn],R[maxn];
void dfs(int k){
    if(k==n+1){
        for(int i=1;i<n;i++)
            printf("%d ",a[i]);
        printf("%d\n",a[n]);
    }
    for(int i=1;i<=n;i++){
        if(!col[k]&&!row[i]&&!L[i-k+n]&&!R[i+k-1]){
            a[k]=i;
            col[k]=1,row[i]=1,L[i-k+n]=1,R[i+k-1]=1;
            dfs(k+1);
            a[k]=0;
            col[k]=0,row[i]=0,L[i-k+n]=0,R[i+k-1]=0;
        }
    }
}
int main()
{
    cin>>n;
    memset(col,0,sizeof(col));
    memset(row,0,sizeof(row));
    memset(L,0,sizeof(L));
    memset(R,0,sizeof(R));
    dfs(1);
    return 0;
}
```



下面为大家推荐1道不错的回溯相关的题目，供大家去练习。

### 迷宫

很标准的一道回溯相关的问题，往四个方向搜索，注意不能搜到自己。希望读者能够在先独立完成的情况下，在参考我的代码。

```
#include "iostream"
#include "cstdio"
#include "cstring"
#include "string"
using namespace std;
const int maxn=15;
int s[maxn][maxn];
int xx[]={0,0,-1,1};
int yy[]={1,-1,0,0};
int n,m,t;
int ans,sx,sy,dx,dy;
```

```
int vis[maxn][maxn];
void dfs(int x,int y){
    //cout<<x<<" "<<y<<endl;
    if(x==dx&&y==dy){
        ans++;
    }else{
        for(int i=0;i<4;i++){
            int nx=x+xx[i],ny=y+yy[i];
            if(nx==sx&&ny==sy)  continue;
            if(nx>=1&&nx<=n&&ny>=1&&ny<=m&&!vis[nx][ny]&&!s[nx][ny]){
                vis[nx][ny]=1;
                dfs(nx,ny);
                vis[nx][ny]=0;
            }
        }
    }
}
int main()
{
    cin>>n>>m>>t;
    cin>>sx>>sy>>dx>>dy;
    ans=0;
    for(int i=0;i<10;i++){
        for(int j=0;j<10;j++)
            s[i][j]=0;
    }
    for(int i=0;i<t;i++){
        int x,y;
        cin>>x>>y;
        s[x][y]=1;
    }
    dfs(sx,sy);
    cout<<ans<<endl;
}
```


光学会了搜索，很可能我们写出来的代码无法通过评测，原因是因为时间复杂度太高，因为裸的搜索我们需要尝试所有的可能性，但其实有些可能性，我们在未计算到底之前就知道它不是最优解，所以对于这种状态，我们就没必要再继续搜索下去，而是直接返回，去搜索其他的状态，这就是我们在搜索当中常用的一种技巧，俗称为“剪枝”。

下面我们通过一个例子来说明一下方法。

## 例2：分解任意一个整数n

格式如下：n=a1*a2*a3*……*ak

比如对于12，就有:

```
12=12
12=6*2
12=4*3
12=3*4
```

```
12=3*2*2
12=2*6
12=2*3*2
12=2*2*3
```

共8种分解方式，现在给你一个数N，让求出总共有多少种分解方式。

分析：许多通过拿着这样一个题一看，就觉得直接暴力求解就好，于是就写出了这样的代码my code 于是自信提交，得到了一个大大的TLE，仔细想想，在这个题目中，很多状态我们是不需要的，于是我们可以这样考虑。首先，我们可以在$\sqrt{n}$的时间内算出n的约数，然后把约数按照从小到达排序，通过约数去求解这个问题，如果发现当前值与约数乘积已经大于n了，就不在继续求解下去，直接进行剪枝即可。

```
#include "iostream"
#include "cstdio"
#include "cstring"
#include "string"
#include "algorithm"
#include "cmath"
using namespace std;
const int maxn=1e5+10;
int cnt,a[maxn],len;
void init(int x){
    int t=int(sqrt(double(x+1)));
    for(int i=2;i<=t;i++){
        if(x%i==0){
            a[++len]=i;
            if(x/i!=i){
                a[++len]=x/i;
            }
        }
    }
}
int n;
void dfs(int d){
    cnt++;
    for(int i=1;i<=len;i++){
        if(d*a[i]>n)  break;
        if(n%(d*a[i])==0&&n!=d*a[i])
            dfs(d*a[i]);
    }
}
int main()
{
    scanf("%d",&n);
    cnt=0,len=0;
    init(n);
    sort(a+1,a+1+len);
    dfs(1);
```

```
        printf("%d\n",cnt);
        return 0;
    }
```

这样交上去，就会得到一个满意的AC。

上述属于比较常见的可行性“剪枝”问题，是一类比较容易理解的“剪枝”问题，也很适合新手，但是“剪枝”作为搜索中最重要的一种技巧，应用远不止于此，但受限于篇幅，笔者在此无法为大家把“剪枝”展开写，就为大家推荐几篇非常不错的博客，供大家学习。

第一个是奇偶剪枝的，这也是一种较为常用的“剪枝”算法，推荐的经典习题是HDU1010，推荐的博客是点我。

第二个是Alpha-Beta剪枝，这一类剪枝在传统算法和现代算法中都有应用，推荐的博客是点我。


前面都是孤立地在讲搜索，下面我们来看看搜索跟其他算法的一些结合，首当其中当属动态规划了，重所周知，记忆化搜索在很大程度上就是动态规划，关于这部分问题，我在上一篇chat中已经谈到过，这里就不在赘述了。下面我们来看看如何利用状态压缩来对搜索进行优化。

题目

分析:这题跟前面全排列那题很像，我们暴力的做法就是求出所有全排列，然后在去找到时间最小的，然而时间复杂度15!，显然是我们无法接受的。这时候我们可以这样思考，对于完成作业1，2，3，无论我们通过什么样的顺序去做，所花费的时间是相同的，唯一不同的只是扣掉的分数，所以我们可以考虑把所有这样相同的状态进行压缩。这样，我们就可以用n个二进制来表示n个作业，如果对应的二进制位为1，表示作业已经完成，如果为0，表示作业尚未完成。这样我们可以把这些状态压缩成同一个状态，并对应一个二进制来表示，维护一下这些状态下的最小得分。那状态转移方程怎么写呢？首先我们考虑一个状态i，它可由状态j转移过来，必然说明了i和j之间除了第k位不同以外，其他位都相同，换言之，i完成了第k项工作，j没有完成，同时二者其他工作完成情况都相同。同时我们维护dp[i]的最小值，以及完成工作的当前时间，和该状态对应的前驱结点即可。

```
#include "iostream"
#include "cstdio"
#include "cstring"
#include "string"
using namespace std;
const int maxn=(1<<15)+10;
const int INF=1<<30;
int T,n;
char s[20][110];
int dead[20],fin[20],dp[maxn],t[maxn],pre[maxn];
void output(int x){
    if(!x) return;
    output(x-(1<<pre[x]));
    printf("%s\n",s[pre[x]]);
}
int main()
{
    scanf("%d",&T);
    while(T--){
        scanf("%d",&n);
        for(int i=0;i<n;i++)
            scanf("%s%d%d",s[i],&dead[i],&fin[i]);
        int bit=1<<n;
        for(int i=1;i<bit;i++){
            dp[i]=INF;
            for(int j=n-1;j>=0;j--){
                int temp=1<<j;
                if(!(i&temp)) continue;
                int num=t[i-temp]+fin[j]-dead[j];
                num=max(0,num);
                if(dp[i]>dp[i-temp]+num){
                    dp[i]=dp[i-temp]+num;
                    t[i]=t[i-temp]+fin[j];
                    pre[i]=j;
                }
            }
        }
```

```
            printf("%d\n",dp[bit-1]);
            output(bit-1);
        }
        return 0;
    }
```

当然，关于状态压缩更多的知识，可以参看这篇文章。



上面谈到的更多是关于DFS的一些知识，下面我们来聊一聊BFS。跟BFS相关的基础知识，在各大数据结构教材当中已经有了比较多的讲解，这里就不在赘述了，不太了解的同学可以参看这里。跟DFS用栈作为底层不同，BFS的底层用队列进行维护，每次进行入队和出队操作。在求图中不带权值的最短路时，用BFS进行扩展，是非常方便的方法。

## 例3：小A和uim之大逃离

分析：这题与普通的BFS不同，有了一次跳变的过程，因此在这里多设置一个变量来标记状态，如果是跳变过去的，标记为1，而移动过去的标记为0。这样，我们就可以像普通的BFS那样来做这题了

```
#include "iostream"
#include "cstdio"
#include "cstring"
#include "queue"
using namespace std;
const int maxn=1100;
int vis[maxn][maxn][2];
int h,w,d,r;
```

```
char s[maxn][maxn];
int dx[]={1,-1,0,0};
int dy[]={0,0,1,-1};
struct Node{
    int x,y,flag,step;
};
int bfs(){
    Node t;
    t.x=1,t.y=1,t.flag=0,t.step=0;
    queue<Node>p;
    vis[t.x][t.y][t.flag]=1;
    p.push(t);
    while(!p.empty()){
        Node e,f;
        e=p.front(); p.pop();
        if((vis[e.x][e.y][0]||vis[e.x][e.y][1])&&(e.x==h&&e.y==w))
        return e.step;
        for(int i=0;i<4;i++){
            int x=e.x+dx[i],y=e.y+dy[i];
            if(x<=0||x>h||y<=0||y>w)  continue;
            if(s[x][y]=='#'||vis[x][y][e.flag])  continue;
            vis[x][y][e.flag]=1;
            f.x=x,f.y=y,f.flag=e.flag,f.step=e.step+1;
            p.push(f);
        }
        if(e.flag==0){
            int xx=e.x+d,yy=e.y+r;
            if(xx<=0||xx>h||yy<=0||yy>w) continue;
            if(s[xx][yy]=='#'||vis[xx][yy][1])  continue;
            vis[xx][yy][1]=1;
            f.x=xx,f.y=yy,f.flag=1,f.step=e.step+1;
            p.push(f);
        }
    }
    return -1;
}
int main()
{
    cin>>h>>w>>d>>r;
    getchar();
    for(int i=1;i<=h;i++){
        for(int j=1;j<=w;j++)
        scanf("%c",&s[i][j]);
        getchar();
    }
    memset(vis,0,sizeof(vis));
    cout<<bfs()<<endl;
}
```



与DFS一样，BFS同样也可以进行”剪枝”,优秀的“剪枝”经常能够达到意想不到的效果，下面来看一道题。

## 题目

分析：这是今年沈阳站的一道题目，如果我们用裸的BFS，将所有最大的值入队列，然后做BFS，复杂度是O(n*L),L表示环的长度，显然是不能过的，但是我们加上一些剪枝效果就大不相同了。(1值小于当前层最大值的点移出队列。(2)同一层在相同位置的移出队列。加上这样的剪枝竟然能够神奇卡过，大雾。

```
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 150000 + 50;
int a[MAXN], D[MAXN], cur[MAXN], vis[MAXN];
char s[MAXN];

struct Node
{
    int v, pos, cur;
    Node() {}
    Node(int v, int pos, int cur) :v(v), pos(pos), cur(cur) {}
};

struct compare
{
    bool operator()(const Node &a, const Node &b) const
    {
        if (a.cur != b.cur) return a.cur > b.cur;
        else if (a.v != b.v) return a.v < b.v;
        return a.pos > b.pos;
    }
};

priority_queue<Node, vector<Node>, compare> pq;

int main()
{
    int t, cas = 1;
    scanf("%d", &t);
    while (t--)
    {
        int n;
        scanf("%d", &n);
        scanf("%s", s);
        printf("Case #%d: ", cas++);
        int maxx = 0;
        for (int i = 0; i < n; ++i)
        {
            cur[i] = -1;
            vis[i] = -1;
            a[i] = s[i] - '0';
            maxx = max(maxx, a[i]);
```

```
        }
        for (int i = 0; i < n; ++i)
        {
            D[i] = int(((long long)i * (long long)i + 1) % (long
long)n);
        }
        for (int i = 0; i < n; ++i)
        {
            if (a[i] == maxx) pq.push(Node(maxx, i, 0));
        }
        while (!pq.empty())
        {
            Node top = pq.top();
            pq.pop();
            if (cur[top.cur] == -1) cur[top.cur] = top.v;
            if (cur[top.cur] > top.v) continue;
            if (vis[top.pos] < top.cur) vis[top.pos] = top.cur;
            else continue;
            if (top.cur == n - 1) continue;
            pq.push(Node(a[D[top.pos]], D[top.pos], top.cur + 1));
        }
        for (int i = 0; i < n; ++i)
        {
            printf("%d", cur[i]);
        }
        printf("\n");
    }
    return 0;
}
```



同样BFS还可以记忆化，下面再来看一道题。

题目

分析:对于这一道题，如果我们对每次查询都进行操作的话，很显然会TLE，对于每一个连通块，我们知道在同一个连通块里的所有格子，每次询问他们，所能到达的格子是相同的，因此，我们对访问过的连痛块进行记忆化，下次再访问的时候直接调用结果即可。

```
#include "iostream"
#include "cstdio"
#include "cstring"
#include "string"
#include "queue"
using namespace std;
const int maxn=1010;
int vis[maxn][maxn],s[maxn][maxn],n,m,ans[maxn*maxn],num;
struct node{int x,y;};
int dx[]={-1,1,0,0};
int dy[]={0,0,-1,1};
```

```
int bfs(int sx,int sy){
    queue<node>que;
    node t;
    t.x=sx,t.y=sy;
    int sum=0;
    sum++,num++;
    vis[sx][sy]=num;
    que.push(t);
    while(!que.empty()){
        node u=que.front();
        que.pop();
        for(int i=0;i<4;i++){
            int nx=u.x+dx[i],ny=u.y+dy[i];
            if(nx<1||nx>n||ny<1||ny>n||vis[nx][ny])  continue;
            if(s[nx][ny]!=s[u.x][u.y]){
                node tt;
                tt.x=nx,tt.y=ny;
                vis[nx][ny]=num;
                sum++;
                que.push(tt);
            }
        }
    }
    ans[num]=sum;
    return sum;
}
int main()
{
    scanf("%d%d",&n,&m);
    for(int i=1;i<=n;i++){
        string str;
        cin>>str;
        for(int j=1;j<=n;j++) s[i][j]=str[j-1]-'0';
    }
    num=0;
    while(m--){
        int x,y;
        scanf("%d%d",&x,&y);
        if(!vis[x][y]){
            cout<<bfs(x,y)<<endl;
        }else{
            cout<<ans[vis[x][y]]<<endl;
        }
    }
    return 0;
```



好了，这次跟大家聊到的搜索就是这么多了，但是搜索算法博大精深，我们这次讲到的只是非常皮毛的东西，要想练好搜索，需要大家平日里多做题，多思考，多总结，才嫩掌握更多的新姿势。下次chat，我会跟大家聊聊一些跟启发式搜索相关的东西，如果大家发现我的文章有什么问题，欢迎大家指出，谢谢大家的支持。