搜索

——吴兆昕

搜索

搜索的本质是一种枚举，枚举所有的可能来找到可能的解或者最优解，在一堆结果中找到所需要的，所以叫做搜索。

搜索与纯粹的枚举（有时也被分类进搜索）不同的是，搜索可以在过程中判断接下来的结果是否为满足要求的，如果接下来的结果全不满足要求，可以直接不搜索下一步的结果（被称为**剪枝**）。

根据枚举方式的不同，搜索大致可以分为两种：

- 深度优先搜索（DFS）
- 广度优先搜索（BFS）

搜索

搜索一般是一个多步的过程（单步的可以直接用循环枚举），这里以枚举一个排列举例，搜索的第一步为枚举这个排列的第一项，第二步为枚举第二项……

深度优先搜索（深搜）是能进下一步就进下一步，比如枚举完第一项为1，立即枚举第二项3，再枚举第三项2，找到一个排列后，再继续第二步未完成的工作，枚举第二项2。
广度优先搜索（广搜）则先枚举当前步所有的可能，如第一步枚举1??、2??、3??，第二步枚举12?、13?、21?、23?、31?、32?，一般广搜将每一步的结果都放在一个队列中，然后每次将队首的下一步结果都放入队列，然后删除队首，直到队列为空。

搜索

搜索的每一步得出的结果叫做**状态**，例如枚举排列时出现的 1?? 就是一个状态，搜索消耗的时间直接与状态总数相关。

广搜一般在状态容易表示（指便于储存），且要求步数最少时使用。例如一个二维平面上的坐标就是一个容易表示的状态，可以直接用`(x,y)`表示，而一个集合的元素，就不太好表示，当然，可以用一个平衡树（`std::set`）表示，但是会消耗大量的空间。

在遇到没有要求最少步数的搜索问题时，一般考虑优先使用深搜，在遇到如要求步数最少的走迷宫这类问题时才使用广搜。

DFS

深搜的基本步骤：

- 检查状态（终态或不合法状态）
- 修改状态
- 递归
- 撤销修改

注意深搜时在终止状态要结束搜索，否则会导致死递归，同时要注意逻辑上是否会出现死递归的情况。
在递归函数的内部，不要开太多变量（包含形参），比如开一个很长的一维数组，本地运行可能会爆栈（运行错误的一种）。

DFS

以下是一个写得不是很优美的搜索，用于枚举排列。

```
int a[11];
int step = 1;
void dfs() {
    if(! check()) return; // 非法状态直接退出 同时起到剪枝的作用
    if(step > 8) { output(); return;} // 找到一个排列
    for(int i = 1; i <= 8; i ++) {
        a[step] = i;
        step ++;
        dfs();
        step --; // 撤销
    }
}
void work() { dfs(); return;}
```

DFS

有时我们可以将状态空间消耗较小的一部分当作参数传递以减少不必要的撤回，并且使代码变得更清晰。

```
int a[11];
void dfs(int step) {
    if(! check(step)) return;
    if(step > 8) { output(); return;}
    for(int i = 1; i <= 8; i ++)
        a[step] = i, dfs(step + 1);
}
void work() { dfs(1); return;}
```

修改 `check` 函数可以实现搜索不同的内容。

剪枝

常用的剪枝有两种：可行性剪枝、最优性剪枝
剪枝配合改变搜索顺序使用更佳

可行性剪枝

如果当前状态非法，或从这一步开始，所有的方法最终都会走向一个非法的状态，那么可以直接剪去当前状态。

最优化剪枝

在解决最优化问题时，从这一步开始，永远都到不了一个比目前最优解更优的状态，可以直接剪枝。

可行性剪枝

一个不恰当的例子：
找出所有第五项为5的长度为5的排列。

在搜索时，如果我们将5放到了第一项，那么我们无论第二、三、四项放什么，最终都是不合法的，这个状态就不可行。

在搜索时，前四步每次检测5有没有被使用，如果用过了就直接`return`，这样可以减少很多搜索量。

当然也可以先把5放在第五项（改变搜索顺序）。

最优化剪枝

一个不恰当的例子：
从5个正数中选3个数使得和最小。

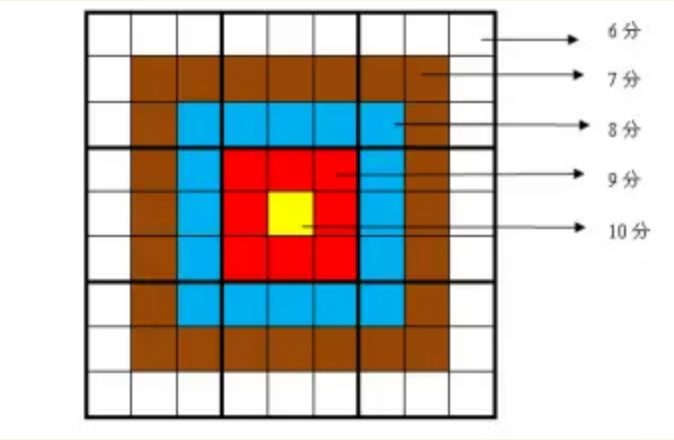
假设我们先搜到了一个和为6的结果，然后我们现在在枚举选的第一个数，并且我们选到了6，那么后面两个数无论我们怎么选，都无法使得和小于6，都无法比之前找到的方案更优，所以我们可以直接剪去这个状态。

当然此题直接选3个最小的即可。

[NOIP2009 提高组] 靶形数独

数独，但是每个格子有对应的分数，最终完成的得分为每个格子对应的分数乘上格子上的数，求最高得分。

最中间的格子分数为10，每往外一圈就减1。



数据范围：非0的格子个数大于等于24个。

[NOIP2009 提高组] 靶形数独

首先要求最大值，除了找出所有可能的方案取最大值外似乎没有什么更好的方法，所以我们直接搜索找出所有方案。

最朴素的搜索，直接从第一排第一个往后扫，遇到一个空位就找一个数填上去，直到全填满，然后算一下分数。

如果每个空位都把1-9试一遍，那么需要枚举9^{51}种情况，显然无法通过此题。

所以我们需要进行剪枝和改变搜索顺序。

[NOIP2009 提高组] 靶形数独

可行性剪枝

如果某一行或某一列或某个九宫格出现了相同的数，那么显然不合法，直接剪掉，或者在填的时候就只填可以填的数。

如果填完某个数，出现了某个位置没有可填的数的情况，那么显然也不合法，直接剪掉。

最优化剪枝

如果在剩下的格子里全填9都无法比最优解更优，直接剪掉。

[NOIP2009 提高组] 靶形数独

改变搜索顺序

有很多种方法，这里列举两个。

每次找到空位最少的一行，然后把这一行填满。

每次找到可填的数最少的格子，先填这一个。

注意，改变搜索顺序并不会直接导致搜索的状态数减少，但可以在剪枝的时候剪去更多不合法的情况，达到加快搜索顺序的效果。

~~数据较水，通过此题不代表代码能通过所有可能的情况~~

BFS

广搜一般只会到达每个状态一次，对每个状态只找到步数最少的方案，在此之后如果再遇到这个状态就直接跳过。

广搜的一般步骤：

- 将初始状态压入队列
- 取队首状态，用队首状态扩展出新的状态压入队列，直到队首为空
- 找到需要的状态时可以直接结束搜索

一般我们使用一个标记来记录某个状态是否被到达过或到达这个状态消耗的步数。

BFS

例：每次可以将一个数加a或减b，加完减完后这个数必须在0到100000之间，问把x变成y最少需要多少步。

状态：x是多少
定义一个`vis`数组来记录把x变成这个数需要的步数，初始全为-1，表示无法到达。

首先将x放入队列，令`vis[x]=1`。
然后每次取队首p，看p+a和p-b有没有被访问过，没访问过就让`vis[p+a]=vis[p]+1`，然后放进队列。

BFS

```
int vis[100001];
queue<int> q;
void bfs(int x, int y) {
    for(int i = 0; i <= 100000; i ++) vis[i] = -1;
    q.push(x), vis[x] = 0;
    while(! q.empty()) {
        int p = q.front();
        q.pop();
        //if(p == y) break;
        if(p + a <= 100000 && vis[p + a] == -1)
            vis[p + a] = vis[p] + 1, q.push(p + a);
        if(p - b >= 0 && vis[p - b] == -1)
            vis[p - b] = vis[p] + 1, q.push(p - b);
    }
}
```

[NOIP2017 普及组] 棋盘

一个$m \times m$的棋盘，每个格子上颜色为无色、红色或黄色，每次能往上下左右四个方向移动一格，移动到同色格子不需要金币，异色格子需要1金币，不能移动到无色格子，但可以用2金币将下一个无色格子的颜色临时变成指定颜色，离开后格子又会变成无色，且魔法不能连续使用。

问从左上角到右下角最少需要多少金币。

$m \leq 100$

有颜色的格子数 ≤ 1000

[NOIP2017 普及组] 棋盘

状态：所在的位置（x,y），是否使用魔法及变的颜色（c）。
可以使用一个结构体保存状态，使用一个三维数组来记录每个状态是否被到达。
与普通广搜不同的是，一步的消耗不为定值，所以一个状态可以被多次访问，在遇到一个更优的方案时，需要将这个状态再次入队，可以再用一个数组记录某个状态是否在队列中来防止重复入队。

```
struct state {
    int x, y; // 坐标
    int c; // 0-未使用魔法 1-变红 2-变黄
};
queue<state> q;
int vis[101][101][3];
```

end