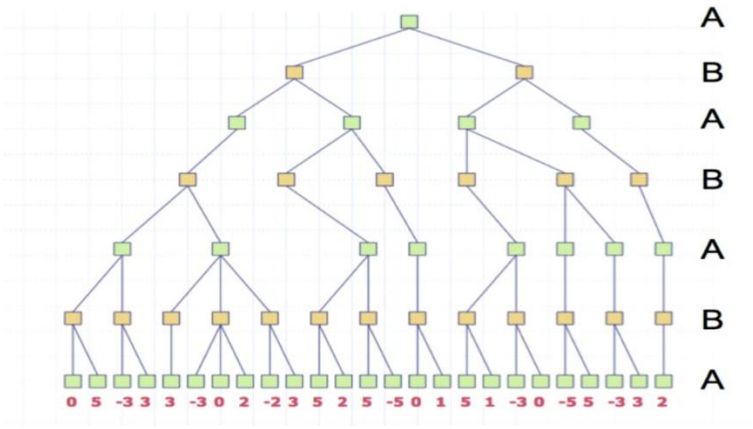
人工智能 hw2

17343023 董宸宇

1. 画出 alpha–beta 剪枝之后的树



在这里令绿色结点为 max 结点，黄色结点为 min 结点

从最左下角沿着右上方向去推，当推到高层的时候再往下推，来确定各个点的上限或下限，当一个点上限小于下限时，它的子结点就可以直接被剪掉

向上推

Min–>max：能确定上层 max 结点的下限，因为 max 结点最小值就是它下层 min 结点的值

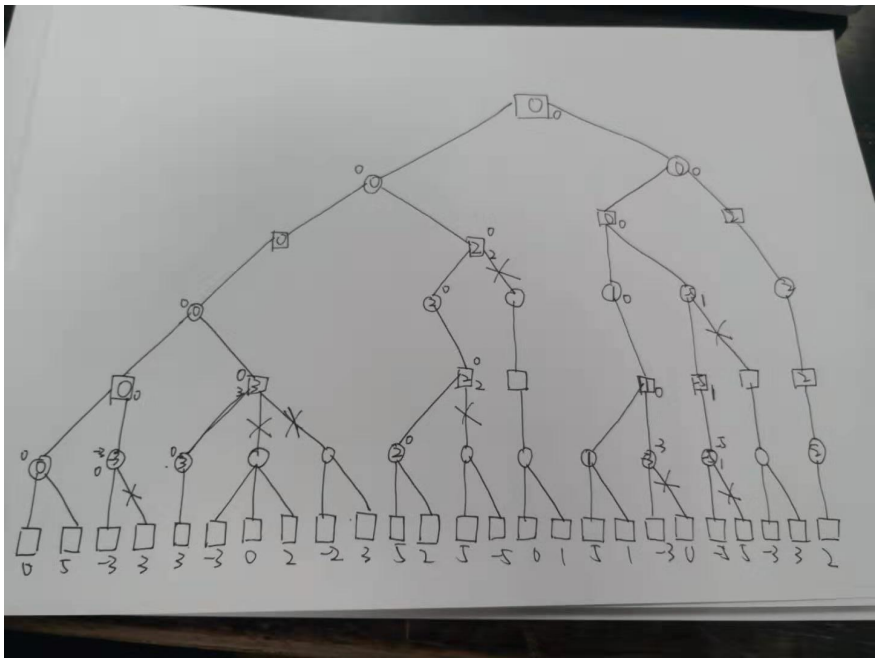
Max–>min：能确定上层 min 结点的上限，因为 min 结点最大值就是它下层 max 结点的值

向下推

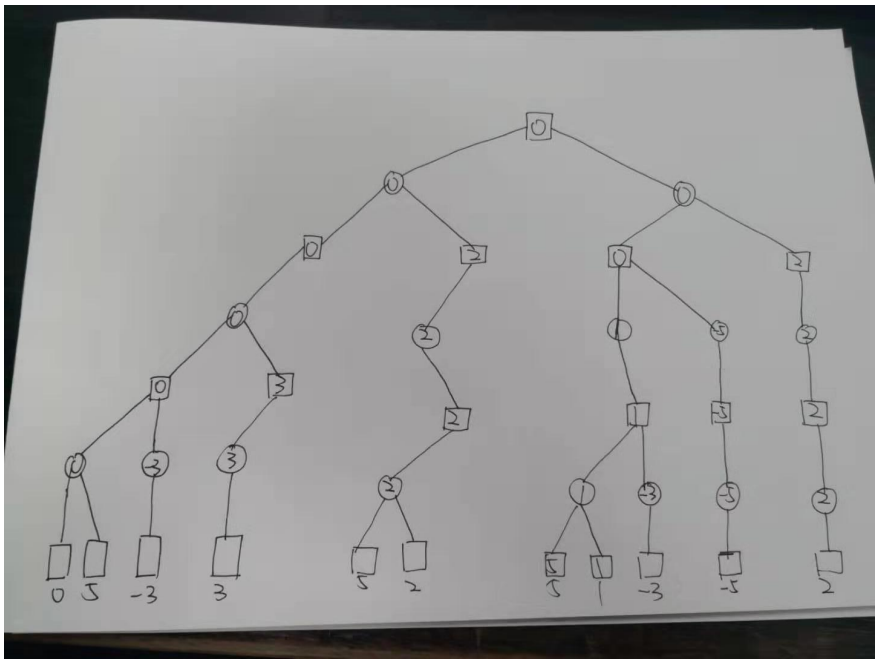
Min–>max：能确定下层 max 结点的下限，因为下层 max 结点的值必须大于或等于上层 min 结点的值

Max–>min：能确定下层 min 结点的上限，因为下层 min 结点的值必须小于等于它上层 max 结点的值

剪枝过程大概如下图



最终结果如下图



2. 数独问题

   题目中要求三种解法

   1. 暴力法
   2. 回溯法
   3. 使用了 MRV(最小剩余值)启发式的前向搜索算法

   暴力法:这个题目纯暴力肯定不现实，假设有 n 个空位，每个空位都有 9 种走法，那么一共就是 9^n 种情况，复杂度太高，所以这里我还是提前对行和列进行了筛选，最后得到答案之后在检测每一个 3*3 的小方格是否满足条件，核心代码如下

检测行和列是否有冲突

```
for(int i=0;i<9;i++){
    if(num[row][i]==key||num[i][column]==key){
        return false;
    }
}
```

检测每个小方格是否都满足条件

```
for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){
        if(num[r][c]==num[row+i][column+j]&&((row+i!=r)||(column+j!=c))){
            return false;
        }
    }
}
```

当然这种算法有很大的改进空间，其实将 3*3 小方格是否发生冲突加入到与行列一起判断即可降低时间复杂度

这样的话就可以一次判定我们的数独在某种状态下各个数字之间是否有冲突

回溯法：

接下来我们在每个位置上都进行 1—9 的尝试，同时利用上面这段代码进行检测，如果这个数字填进去之后不会引起冲突，则我们可以尝试把数字填到方格里面，同时进入更深一层的搜索，随着搜索的不断深入，可能会出现两个结果，成功(81 个方格全部填满)/失败(选择的是一条死路，填不下去了)，如果成功，则需要逐层返回，如果失败，则要将之前填好的数字全部清 0，核心代码如下

```
for(int i=1;i<=9;i++){
    if(Check(n,i)){
        num[row][column]=i;
        DFS(n+1);
       if(sign) return;
        num[row][column]=0;
    }
}
```

Check：即上面提到的检验函数，两个参数 n，i，n 代表当前检验的位置，i 表示在这个位置上的值

位置排列

0 1 2 3 4 5 6 7 8

……

72 73 74 75 76 77 78 79 80

num：二维数组，用来表示数独

sign：是一个标记变量，当 81 个格子都被填满之后 sign 变为 true，在上面这段代码中如果 sign 变成 true 的话就会逐层返回

DFS：用于搜索的函数

使用了 MRV(最小剩余值)启发式的前向搜索算法：

这个算法其实跟回溯法有些类似，也是在每一个还没有填上数字的地方去试，但具体操

作上有一些区别，主要区别有两点
传统的回溯法是从头到尾按照顺序去试，并且从 1 到 9 都去试一遍，
不过数独有一个特性，当在某个位置上添了一个数字之后，对该位置上对应的行，对应的列，以及对应的 3*3 小方格都会有限制，因为在它们占据的位置上再也不能出现这个数字了，所以这可以缩小我们的搜索范围，所以我们可以将 81 个点中每一个空白的结点都附带一个表，在这里可以使用 vector 来表示，每当填入了一个数字的时候，那填了数字的位置所在的行，列，3*3 小方格里面的方格所对应的 vector 都要删除这个被填入的元素，
所以我们对每个位置，不必从 1—9 试一遍，只需要尝试其对应的 vector 里面的元素即可
当我们了解了每个位置有几个可能的候选值之后，我们还可以对算法在回溯法的基础上进行一下优化，传统的回溯法是按照位置顺序来进行填充，在这里的启发式中，我们对填充的方格的顺序也进行调整，在这里我们优先选择候选值最少的位置，这样可以更快的确定某个位置的值，当确定的位置多起来，那么整个数独对其他为填充位置的限制也就多了起来，这样就起到了剪枝的作用，如果把我们的填充过程比作一棵树的话，我们这种做法就相当于让分支少的结点尽可能充当了父结点，从而使得需要搜索的分支变得更少
体现在程序里面有以下几个关键函数
candinum(int a[][],int row,int column)：记录位置为(row,column)的候选值的数目
min_choice()：选出当前候选值最少的结点，返回该结点的位置
delete_num(int index,int num)：在位置为 index 的方格候选值里面删除 num
代码在这里就先不贴了，提交的时候会带上

复杂度：
纯穷举法的时间复杂度是 9^n，空间复杂度为 On
回溯法的时间复杂度很难具体求出来，但是递归调用次数 9*9!,具体取决于数独里面空位的个数,空间复杂度为 On
基于 MRV 的前向搜索的时间复杂度跟上面类似，递归调用次数 m*9!(m 为所有位置平均的候选值个数)，空间复杂度也有点难以具体计算，不过大体上为 mn

测试结果：
在此列举回溯与基于 MRV 的前向检索
回溯：

```
3 7 2 1 8 6 9 5 4
1 8 5 3 4 9 7 2 6
6 9 4 2 5 7 1 8 3
9 4 8 7 1 3 5 6 2
5 1 6 9 2 4 3 7 8
2 3 7 5 6 8 4 1 9
4 6 3 8 7 5 2 9 1
8 5 1 4 9 2 6 3 7
7 2 9 6 3 1 8 4 5

62/1000000 (s)
```

```
5 3 2 1 7 8 6 9 4
7 6 1 3 4 9 5 2 8
4 8 9 5 2 6 7 1 3
9 4 5 2 1 3 8 6 7
8 2 3 4 6 7 9 5 1
1 7 6 8 9 5 3 4 2
2 5 7 9 3 1 4 8 6
3 9 4 6 8 2 1 7 5
6 1 8 7 5 4 2 3 9

215/1000000 (s)
```

```
1 3 2 9 7 8 6 4 5
5 9 7 6 2 4 8 1 3
6 4 8 1 5 3 2 9 7
2 5 1 7 3 9 4 6 8
4 8 6 2 1 5 3 7 9
3 7 9 4 8 6 1 5 2
7 2 5 3 6 1 9 8 4
8 1 4 5 9 2 7 3 6
9 6 3 8 4 7 5 2 1

354/1000000 (s)
```

```
3 2 6 5 7 9 1 8 4
4 7 9 1 3 8 2 5 6
5 8 1 4 2 6 7 3 9
1 3 5 2 6 4 9 7 8
9 6 2 7 8 5 3 4 1
8 4 7 3 9 1 5 6 2
7 5 4 8 1 2 6 9 3
2 9 8 6 5 3 4 1 7
6 1 3 9 4 7 8 2 5

147/1000000 (s)
```

```
6 1 3 8 4 5 9 2 7
2 4 9 1 6 7 3 5 8
5 7 8 2 9 3 1 4 6
3 5 4 9 7 2 8 6 1
8 2 6 3 1 4 7 9 5
7 9 1 6 5 8 4 3 2
1 6 2 4 8 9 5 7 3
4 3 5 7 2 1 6 8 9
9 8 7 5 3 6 2 1 4

103/1000000 (s)
```

基于 MRV 的前向检索

```
3 7 2 1 8 6 9 5 4
1 8 5 3 4 9 7 2 6
6 9 4 2 5 7 1 8 3
9 4 8 7 1 3 5 6 2
5 1 6 9 2 4 3 7 8
2 3 7 5 6 8 4 1 9
4 6 3 8 7 5 2 9 1
8 5 1 4 9 2 6 3 7
7 2 9 6 3 1 8 4 5
227/1000000 (s)
```

```
5 3 8 1 7 2 6 9 4
7 6 1 3 4 9 5 2 8
2 4 9 5 8 6 7 1 3
4 9 5 2 1 3 8 6 7
8 2 3 4 6 7 9 5 1
1 7 6 8 9 5 3 4 2
3 5 7 9 2 1 4 8 6
9 8 2 6 3 4 1 7 5
6 1 4 7 5 8 2 3 9
314/1000000 (s)
```

```
6 5 2 3 9 8 1 4 7
1 9 4 6 7 2 8 5 3
3 8 7 1 5 4 6 9 2
2 1 8 7 3 9 4 6 5
4 3 6 2 1 5 9 7 8
5 7 9 4 8 6 2 3 1
7 2 5 9 4 1 3 8 6
8 4 1 5 6 3 7 2 9
9 6 3 8 2 7 5 1 4
764/1000000 (s)
```

```
3 6 2 5 7 9 1 8 4
7 4 9 3 1 8 2 5 6
5 8 1 4 2 6 7 3 9
8 2 6 1 4 3 9 7 5
9 1 3 7 8 5 6 4 2
4 5 7 6 9 2 3 1 8
2 7 5 8 6 1 4 9 3
1 9 8 2 3 4 5 6 7
6 3 4 9 5 7 8 2 1
455/1000000 (s)
```

```
6 7 8 1 4 3 5 2 9
3 4 9 2 6 5 8 7 1
2 1 5 8 9 7 3 4 6
5 9 1 3 7 2 4 6 8
8 6 3 9 5 4 7 1 2
7 2 4 6 8 1 9 3 5
1 5 2 4 3 8 6 9 7
4 8 6 7 1 9 2 5 3
9 3 7 5 2 6 1 8 4
433/1000000 (s)
```

有两点值得一提的地方，一是对于一些解很多的数独，回溯法与基于 MRV 的前向检索所得出来的答案会不一样，因为这两种方法的搜索策略就不一样，二是基于 MRV 的前向搜索在时间复杂度上并没有优于回溯法，可能是因为在前向搜索中我们进行的其余操作花费了很多时间，如挑选最小值，删除候选值等，因为我们在这里每填充一个值就要重新进行上述两种操作，所以花费了很多时间