

人工智能基础 搜索大作业 ——火柴棍移动

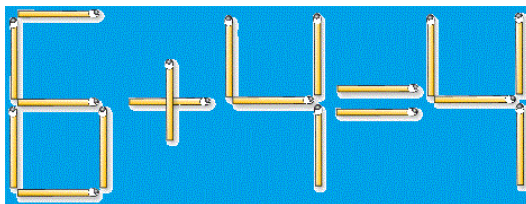
班级：自 71

姓名：屈晨迪

学号：2017010928

1 任务描述

使用火柴棍可以搭出如下图所示的等式，编写搜索算法程序，移动一根火柴使等式成立。



主要任务如下：

- (1) 允许在一个固定的等式库（两位数以内的加减乘法）中选择，从而给出答案；
- (2) 允许使用者自己定义，或者输入一个可以求解的等式。如果无解，回答无解；
- (3) 给出更多的题目和答案；
- (4) (选) 允许移动 2 根火柴棍；
- (5) (选) 给出从等式变为新的等式的题目和难度。

2 问题建模

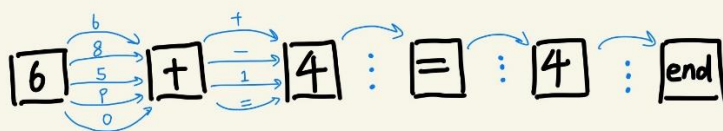
2.1 状态空间

对于本次火柴棍移动问题（必做），参照状态空间表示法给出以下定义：

- ①状态：等式 x 中每个字符移动一根火柴（加、减、自身移位）可能变成的所有等式
- ②初始状态：原始等式 x
- ③目标状态：由原等式 x 移动一根且成立的等式
- ④转换操作：按照移动一根的规则更换等式中某个字符，获得新的等式
- ⑤代价函数：生成一个新等式移动的火柴根数（或移动操作总权值）

选做中移动两根与上述类似：

考虑一个等式 x ，假设 x 由五个字符组成（如 $6+4=4$ ），将每个字符看作一个格点，格点之间有路径相连，每条路径包含上一格点字符可能变成的新字符，如下图所示，搜索时遍历从起始字符到末位字符的所有路径，找出使等式成立且仅移动了一根火柴的全部路径。



3 算法设计和实现

3.1 数据组织结构

算法的实现依托于数据结构的组织，首先介绍本程序中的数据组织。

为了提升搜索效率，程序开始时，调用函数`void getdata_one()`，将等式中可能出现的所有字符（'0-9' '+' '-' '*' '='）移动一根火柴棍能变成的新字符存入一张表中，后续搜索只需要直接从表中查找即可。

整张表为结构体类型的二维数组，定义结构体`match_change`，包含移动权值 `flag` 和变成的字符 `data`，移动权值的复制规则见表 1。则二维数组`mcg[m][n]`中每一个元素都具有一个 `flag` 值和一个 `data` 字符，`m` 对应原始字符，`n` 对应原始字符变成的新字符序号，便于搜索算法的实现。

```
01. struct match_change
02. {
03.     internal int flag;
04.     internal char data;
05. };
```

表 1 移动权值赋值规则

| 移动操作 | 不变 | 自身移动一根 | 加一根 | 减一根 |
|--------|----|--------|-----|-----|
| Flag 值 | 0 | 1000 | 10 | 100 |

3.2 搜索算法

搜索算法是这次编程作业的核心部分，整体搜索流程如图 1 所示。本项目中笔者共设计和实现了两种搜索算法——暴力搜索（for 循环）和深度优先搜索（递归），两种算法本质上相似，但在表示和具体实现上又有所不同，搜索效率也有一定差异，分别讨论如下¹。

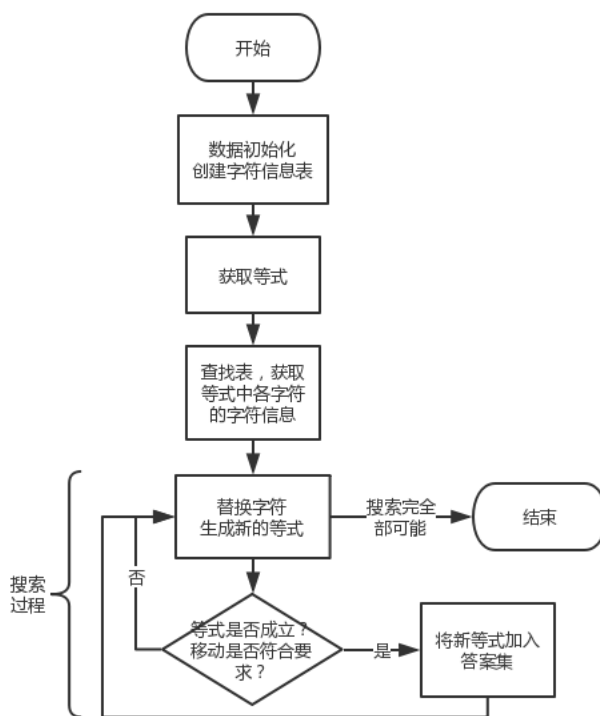


图 1 整体搜索流程

¹ 下文的讨论针对移动一根的基本任务展开，选做中移动两根的情况与其主体思路相似，若有不同之处会专门指出。

3.2.1 暴力搜索 (for 循环)

(1) 基本思路

For 循环搜索的思路很简单, 假设程序获取一个包含 m 个字符的等式 X , 即从第一个字符 c_0 开始, 找到该字符移动一根 (加一根、减一根、自身移动一根) 火柴所能变成的所有字符 $\{c_{00}, c_{01}, c_{02} \dots\}$, 用 c_{00} 替换原等式 X 中的 c_0 , 同时按照 c_{00} 相对 c_0 的移动类型改变等式移动操作的总权值, 再对等式 X 的第二个字符 c_1 进行同样的操作。反复上述步骤直到将最后一个字符 c_{m-1} 替换成 $c_{(m-1)0}$, 获得新等式 X_1 , 此时判断 X_1 是否成立, 并根据移动操作总权值判断等式是否包含非法移动 (移动火柴数量超过一根), 若满足要求, 将该等式 X_1 加入答案集。接着选取最后一个字符可能变成的第二个新字符 $c_{(m-1)1}$, 判断等式 X_2 是否满足要求, 以此类推。

使用 for 循环遍历等式中所有字符可能变成的全部新字符, 获得所有可能的组合等式, 找到其中满足成立和移动规则要求的组成答案集。

从上述描述我们可以看到, 此种方法搜索效率极低, 要遍历等式中所有字符, 意味着等式包含几个字符, 就要嵌套几层 for 循环, 对于上述示例, 时间复杂度为 $O(n^m)$ 。

(2) 算法优化

事实上, 在上述搜索方法中, 我们获得了大量不符合移动规则的新等式, 而有些在没有获得完整新等式时已经可以判断是否符合要求。如当等式前两个字符替换完成时, 发现出现了两个字符各自加/减一根, 或各自移动了一根火柴, 此时移动操作已经非法, 后续的搜索可以舍弃。

因此, 我们有两种思路可以选择, 一是在 for 循环中边遍历边评估移动操作权值, 当权值超过或明显不符合要求时, 直接跳出该次循环; 二是分类讨论, 即本项目中使用的策略, 移动一根火柴的全部可能包括: ①等式中某一个字符自身移动一根 ②等式中一个字符加一根, 另一个字符减少一根; 移动两根火柴包含的可能性较多, 可以按照两根火柴均不改变所在字符位置、一根改变、两根均换字符位置三种情况思考。

分类讨论结合了问题实际, 将多层 for 循环拆开, 可以减少循环层数, 对效率的提升是明显的。

(3) 算法伪代码

```
//for 循环搜索
void solve_for()
{
    //获取等式中字符信息
    for(int i=0; i<字符串长度; i++)
    {
        //等式字符数字化
        '0-9' -> 0-9;
        '+' -> 10; '-' -> 11; '*' -> 12; '=' -> 13;
        //获取各字符可能变成的新字符个数
    }
    //等式中一个字符自身移动一根火柴
    //等式中一个字符加一根, 一个字符减一根
}
```

3.2.2 深度优先搜索

(1) 基本思路

深度优先搜索算法基本思路与 for 循环基本相同，dfs 使用递归思想，包含搜索和回溯两个主要步骤。

将等式 X 中每个字符看作一个格点，每个字符通过一根火柴的移动操作能变成 n 个新字符，对应于每个格点有 n 条路径，设置数组 *string_flag*[] 标记某格点是否走过，设置字符串 *str_route* 记录经过先前字符替换后的当前等式，设置整型 *flag* 记录整个等式的总移动权值。

从第一个字符 c_0 开始，替换成 c_0 能变成的第一个新字符 c_{00} ，令 *string_flag*[0] = 1，标记第一个格点已走过，更新字符串 *str_route*，更新总移动权值，步入下一格点，每次判断是否搜索完最后一个字符格点，若搜索完成，判断新获得的等式是否成立，判断整个等式的总移动权值是否符合要求，在移动一根的情况下，应有 *flag* == 1000 || *flag* == 110，分别对应某字符自身移动一根和两个字符一个加一根一个减一根两种情况。进行回溯时，令 *string_flag*[pos] = 0，取消已走过的标记，*flag* 减去上次字符替换的移动操作权值。

(2) 算法伪代码

```
//dfs 递归搜索
bool solvedfs (int pos)
{
    if (搜索完最后一个字符)
    {
        if (总移动权值符合要求 && 等式成立)
        {
            return true;
        }
        else { return false; }
    }
    //获取当前位置 pos 的字符信息
    //等式字符数字化
    '0-9' -> 0-9;
    '+' -> 10; '-' -> 11; '*' -> 12; '=' -> 13;
    //获取当前字符可能变成的新字符个数
    标记 string_flag[pos] = 1，已走过
    For(i=0; i<当前字符能变成的新字符个数; i++)
    {
        //替换字符
        //更新总移动权值
        if (solvedfs(pos+1)) {} //递归
    }
    //更新移动权值
    return false;
}
```

3.2.3 算法性能对比

笔者考虑了暴力搜索、DFS、BFS 三种搜索算法，并在程序中实现了前两种，对于 BFS，笔者认为其消耗存储空间过大，对算法性能提升无多意义，且 BFS 适用于判断是否存在通路并寻找到最短路径等方面，与本项目适配性不高，故不予使用。

表 2 算法性能对比

| | 暴力搜索 | DFS | BFS |
|---------|--|--------------------------|----------------------------|
| 基本思路 | 从遍历等式中字符，用可能变成的字符替换，判断新等式是否成立，是否符合移动规则要求 | | 不断向下一个字符层扩展，最终从所有可能等式中挑出答案 |
| 实现方式 | For 循环 | 递归 | While 循环 |
| 代码简洁程度 | 较繁琐 | 较简洁 | 较简洁 |
| 优化方法 | 分类讨论 | 加入权值评价 | |
| 搜索时间[1] | 较慢，对不同长度的等式搜索时间差异不大 | 较快，且字符少的等式搜索时间明显短于字符多的等式 | 最快，只需遍历等式一次，但可能出现申请空间用时较长 |
| 占用空间 | 小 | 小 | 大，每出现一种组合情况即需要申请一个等式存储空间 |
| 适用性 | 适用 | 适用 | 不适用(更适于寻找最短路径等方面) |

注[1]：调用 C#类stopwatch的函数start(),stop()测试指定程序模块运行时间，两个测试案例如图 2 所示，可以看到对于两个等式 $7 + 3 = 9$ 和 $45 + 46 = 99$ ，使用 for 循环用时均在 0.0044 秒左右，而使用 DFS 递归搜索则与等式长短相关，一个为 0.0007 秒，一个为 0.0026 秒，且均小于 for 循环用时。

file:///C:/Users/12516/Documents/Visual Studio 20

file:///C:/Users/12516/Documents/Visual Studio 20

```
7+3=9
原等式为: 7+3=9
等式成立7+2=9
for循环用时00:00:00.0043084
7+2=9
等式成立7+2=9
递归用时00:00:00.0007462
7+2=9

45+46=99
原等式为: 45+46=99
等式成立49+46=95
等式成立45+48=93
for循环用时00:00:00.0045565
49+46=95
45+48=93
等式成立45+48=93
等式成立49+46=95
递归用时00:00:00.0026207
45+48=93
49+46=95
```

图 2 算法搜索时间测试案例

3.3 等式处理算法

等式处理是本程序中一块主要内容，包含等式成立性判断、等式拆分、求取等式中某字

符个数、更换等式特定位置字符等功能函数，其中以函数`int ifcorrect(string str)`最为主要，该函数用于判断输入的字符串等式是否成立，若成立返回 0，不成立返回 1，具体流程图如图 3 所示。

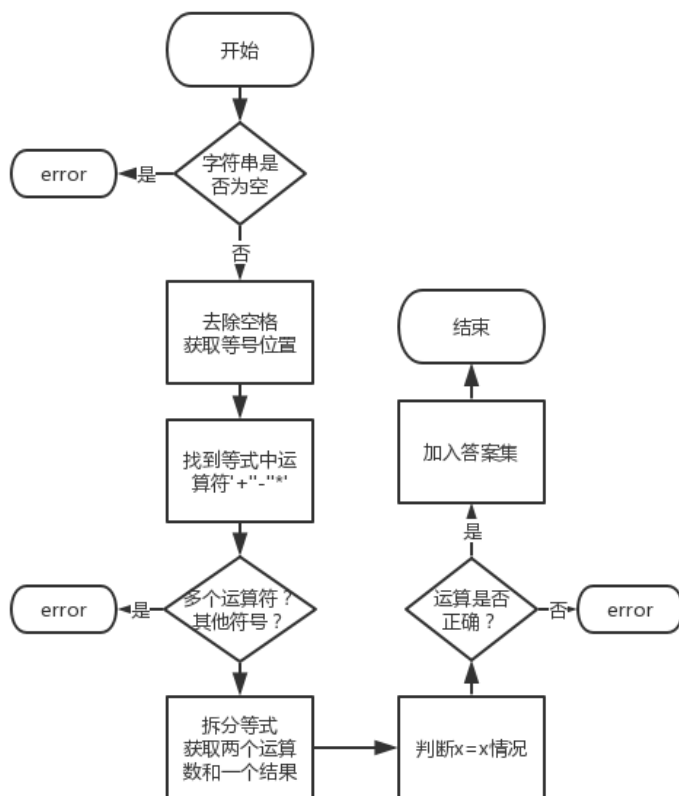


图 3 等式正确性判断流程

3.4 数据库类

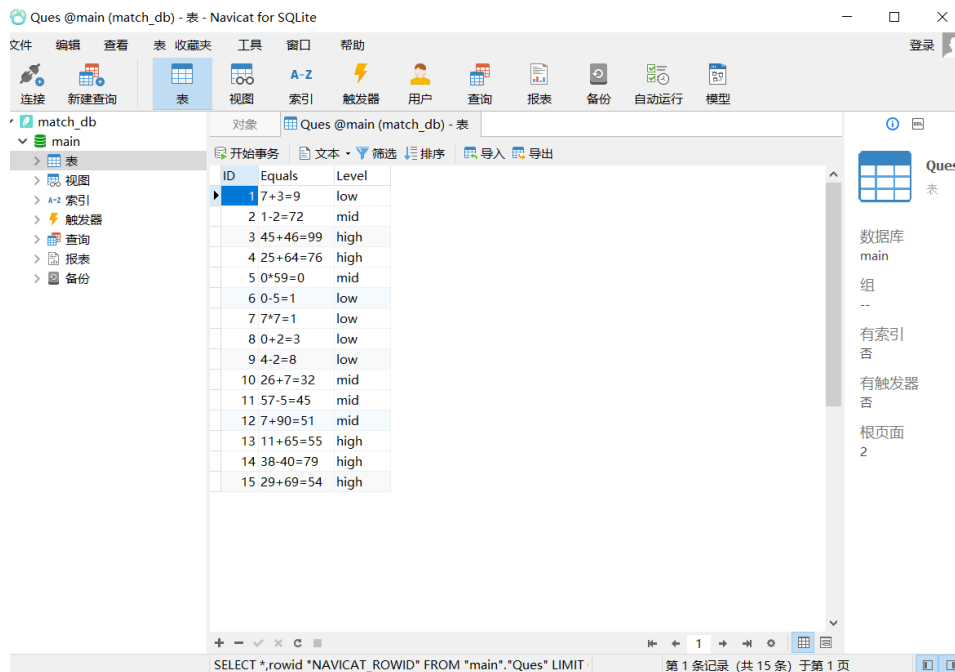
考虑到题目要求从特定等式库中出题，为了便于等式库的创建、增补和删改，将程序与 SQLite 数据库相连，编写 `DataBase` 类，用于数据库的新建、连接、读取等操作，该类中包含的函数如图 4 所示，使用图形界面软件 Navicat for SQLite 可以更便捷地对数据库进行操作，如图 5 为本程序的等式库。

笔者最初也曾尝试将等式库创建为数组形式，但考虑到本程序作为一个游戏项目，应具有处理大量数据的能力，最终选择了接入数据库，其他游戏开发者或玩家可以自行向数据库中导入大批量的等式用于游戏，而不必重读、修改繁杂的代码，程序实用性、扩展性更强。

```

01. //检查数据库是否存在
02. public void CheckDB(string dbPath)
03. //删除数据库
04. public void DeleteDB(string dbPath)
05. //创建表
06. public void Create_An_Table(string dbPath)
07. //删除表
08. public void DeleteTable(string dbPath, string tableName)
09. //向表中插入数据
10. public void Insert(string dbPath, string ans)
11. //从表中读取数据
12. public void ShowData(string dbPath, string tableName, string data)
  
```

图 4 数据库(`DataBase`)类



| ID | Equals | Level |
|----|----------|-------|
| 1 | 7+3=9 | low |
| 2 | 1-2=72 | mid |
| 3 | 45+46=99 | high |
| 4 | 25+64=76 | high |
| 5 | 0*59=0 | mid |
| 6 | 0-5=1 | low |
| 7 | 7*7=1 | low |
| 8 | 0+2=3 | low |
| 9 | 4-2=8 | low |
| 10 | 26+7=32 | mid |
| 11 | 57-5=45 | mid |
| 12 | 7+90=51 | mid |
| 13 | 11+65=55 | high |
| 14 | 38-40=79 | high |
| 15 | 29+69=54 | high |

图 5 特定等式库

4UI 设计和使用说明

4.1 编译和运行环境

本程序为在 VS2012 平台上使用 C# 语言编写的 Windows 应用程序，编译和运行于 .NET Framework 4.5.

4.2 运行方式

Windows 系统下双击文件 *match* 的 *bin* 目录中 *Debug* 下的 *match.exe* 即可直接运行。

4.2 界面设计

(1) 开始界面

开始界面如图 6 所示，点击按钮“开始游戏”进入游戏界面，点击“游戏玩法”查看游戏玩法说明，点击“退出”关闭程序。



图 6 开始界面

(2) 游戏主界面

游戏主界面如图 7 所示。

上方为游戏模式选择按钮，可以选择从题库出题、玩家自主出题、程序随机出题三种模式；右上方为两个下拉菜单，难度等级可以选择低、中、高，难度等级由等式复杂程度确定，移动火柴棍根数可以选择移动一根或两根。

当玩家进入游戏时，游戏默认为随机出题模式，此时问题输入框封锁，火柴棍图片展示低难度的第一道等式题目，玩家可以选择使用 for 循环搜索或递归搜索，选好后，点击“求解”按钮，答案展示在下方文本框中，同时显示答案等式的火柴棍图片。



图 7 游戏主界面

选取一道题库中的高难度题目，求解结果如图 8 所示，答案栏旁边显示出解的个数以及求解搜索算法所用时间，可以供玩家对比不同搜索算法的运行效率，点击“下一解”按钮显示下一条答案，若展示完全部答案，弹出消息提示“已是最后一条”，点击“下一题”，界面清空上道题目的全部有关信息，展示下一道题目。



图 8 题库出题模式下求解界面

点击“自主出题”按钮进入自主出题界面，等式输入框解锁，难度等级下拉菜单封闭，用户可以在文本框中输入两位正整数的加减乘运算等式，当输入非法等式，如等式长度超限、无等号、包含非法字符等情况，会弹出消息提示框，请用户输入合法等式。后续求解过程与题库出题相似。

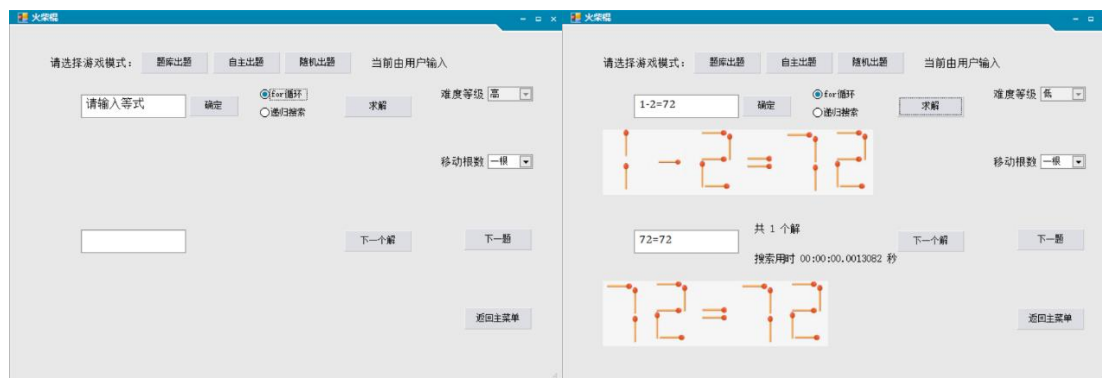


图 9 自主出题模式下的输入求解

随机出题为程序根据用户选择的难度等级随机生成两位正整数的加减乘运算等式，求解过程及界面与前两类相似，此处不做赘述。

5 实验总结

5.1 项目亮点

(1) 功能齐全，界面美观

本程序实现了作业要求中的全部必做和选做要求，提供了题库出题、自主出题和随机出题三种出题模式，游戏过程中，玩家可以自行选择、调整适合自己的题目难度以及移动火柴的根数，并且在移动一根的情况下提供 for 循环搜索和 dfs 递归搜索两种搜索方式的选择，结合下方的搜索用时，让玩家更好的认识两种算法的搜索效率，丰富了程序的交互体验。

另外，使用 C# 工具 `skinEngine` 对界面进行整体美化，在尝试了多种界面皮肤后选择了 `Deepcyan` 皮肤，整体统一的界面风格让使用者有更优的感官体验。

(2) 连接数据库

为了满足从特定等式库出题的要求，为程序接入 `SQLite` 数据库，数据库的引入有助于大量数据的处理，玩家可以按照自身需求自行向数据库 `match_db.db` 中批量导入等式，在游戏中呈现并求解，更加人性化；程序开发者结合数据库相关的 GUI 软件也可以更加便捷的增补、删改等式库。

5.2 存在的问题及可改进之处

(1) 等式局限性

本程序限于两位正整数的加减乘运算，如有需要，可以考虑加入负数运算、除法运算等，这些均是等式处理方面的问题，未涉及算法核心，这里仅给出思路。

要实现负数运算，需要在等式运算部分开头和等式计算结果左侧各加入一个空格符，用于负数的出现，并在字符信息表中添加新字符空格 ' '，可以通过增加一根火柴变成 '-'，增加两根火柴变成 '*' '1' '+' '='，添加完成后运行程序即可；除法运算与加减乘类似，在等式成立性判断函数中加入除法部分，并同样要添加新字符 '/'，可以移动一根变为 '*'，移动两根变为 '1' '+' '='，删去两根变为 ' '。

(2) 移动两根火柴的 dfs 算法实现

在初版程序的编写时，笔者仅考虑了暴力搜索的方法，即使用 `for` 循环，后续优化时想到了可以尝试 `dfs` 的递归算法，于是在移动一根的情况下做了丰富，提供了两种算法设计，但对于移动两根的情况，使用 `dfs` 需要重新编写使用二维数组架构的字符信息表，工作较为繁复，且对移动操作权值的定义也要添加数项，于是未予以实现，其思路、代码与移动一根基本相似，只需改变规范移动操作的判定条件即可。

5.3 心得体会

本次大作业前前后后花费了我两周左右的时间，期间收获良多。本项目的全部程序使用 C# 语言编写，我从头学习了 C# 的使用和 Windows 窗体的设计，现如今已基本掌握了这一门语言。在一开始我选择了七巧板题目，自学了 C#+`openCV` 的相关操作，但最终由于数图知识的欠缺，还是没能成功完成，于是中途转向了火柴棍移动，这个题目的思维难度没有很大，但编程中如果不够细心，会出现一些细节错误，如字符信息表不够完全、等式判断有疏漏等问题，影响最终结果。在完成了暴力搜索算法的编写后，我深感这种算法与未能很好的展示搜索的特质，如状态空间思想的体现，于是添加了深度优先搜索的算法，学完数据结构课程半年多了，重新尝试函数递归还是出了一些问题，最终努力 `de` 掉了其中的 `bug`，感觉这样的算法更能触及搜索的核心。

最后，感谢老师和助教在本次大作业中给予我的帮助与指导！

6 参考文献

[1] <https://blog.csdn.net/u010774394/article/details/38558287>