人工智能基础课程实验报告 1

姓名：章东泉　　**学号：**PB17121706

写在题前：

两个实验都不是直接输到文件里，是在终端打印了相关信息之后复制到.txt 的。

问题一　数码问题

Part1 写在前面

使用的包有：

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string>
#include<iostream>
```

在文件第六行**#define inputfile "../input/2.txt"**，请在测试时事先修改文件路径。Windows 下直接编译运行 A_star_search.cpp 和 IDA_star.cpp 文件即可。

Part2 启发式函数

.启发式函数的主要构建从熟悉的曼哈顿距离开始，我们已知的曼哈顿距离是八数码问题一个一致的启发式，但是效果较差，所以加入了以下两个部分：

1. 行、列的逆序对对数，用于放大状态之间的启发式差异；

2. 在曼哈顿距离的基础上对“7”块的权重进行放大，诱导程序搜索能够移动“7”的状态。

下面给出具体的启发式设计：

4*逆序对对数 + 曼哈顿距离 + “7”的诱导

“7”的诱导设计比较简单，在基础的曼哈顿距离上放大权重，例如，在不移动“7”的操作的上将曼哈顿距离乘 18，在移动“7”的操作上将距离乘 3。具体的数据是数次调整参数的结果。

整个启发式设计是基于曼哈顿距离启发式，并且每个状态的启发式值不小于其曼哈顿距离，故知道启发式是一致的。

Part3 程序结构和算法分析

首先是数据结构层面的设计：

```
struct State{
    int status[5][5];
    int blank1[2];
    int blank2[2];
    int forward_cost;
    int manhattan;
    int disorder;
    int heuristics;
    int award7;
    int value;
    std::string path;
};
```

以上是定义了“状态”的部分。status[][]用于存放数码盘的具体状态，用于后续做状态检查。blank1[]与 blank2[]用于存放两个“0”块的位置，数组第一位和第二位分别表示行、列坐标。forward_cost 存放从初始状态起移动的代价数，每次操作(包括移动数码块“7”）的代价是 1。disorder,manhattan 和 award7 分别存放 Part1 中说明的启发式 3 部分，value 是最终的启发式值。String 类 path 存放自初始状态

的具体操作，从父节点继承。

Open-list 直接组织成链表的形式，每次插入新结点的时候从头节点扫描，保证根据耗散值 value 是有序增加的。理论上应该维护成二叉堆的形式，可以在 O(logn)内完成节点插入，但是这里图省事直接用 O(n)的算法完成了。

Close-list 组织成二叉树的形式，按照启发式的值进行排序，在进行重复状态的检查时，如果检查到一样的启发式值，则深入对数码盘的状态比较，否则启发式值不同，状态一定不同。这里图省事并没有组织成 AVL 的形式（请助教不要打我）。在做 IDA*的时候发现还是用二叉堆比较好，但是用数组方式组织担心空间不够，用树的话虽然避免了空间问题，但是 IDA*每次迭代需要释放之前的 Close-list，在树结构下所有节点的内存释放变成了很麻烦的事情。

以下是自定义函数部分：

```
int checkState(CloseNode* q, Node* p)    //search_Node()调用，比较具体状态
int search_Node(CloseNode* q, Node* p) //重复状态检测
void insert_Tree(CloseNode* q, Node* p) //向 Close-list 插入新结点
void insert_Node(Node* p)                  //向 Open-list 插入新结点
int Manhattan, seven_award, Disorder()//Heuristics_Do 调用，分别计算启发式
                                          //的三个部分
void Heuristics_Do(Node* p)              //计算启发式值
void A_star_Search()                       //A*搜索的主体，循环调用
```

部分伪代码如下：

```
while(Open-list != NULL):
    if is goal state{
        Do print and return;
    }
    for every possible movement{
        if not in Close-list{
            insert into Open-list;
        }
```

```
}
free first node of Open-list;
```

Part4 部分实验结果

具体实验结果见 output/文件夹。

对于测试样例 1，搜索了 261 个节点找到解，共 24 步，约 600ms，对于测试样例 2，搜索了 18 个节点找到解，共 14 步，约 470ms。测试样例 3 的运行时间就...略长，也很吃内存。

在迭代 A*搜索算法上，测试样例 1 搜索了 561 个节点，样例 2 则与 A*搜索一致，时间都较 A*更短，其中样例 1 快了近一倍。第一次搜索时将 Open-list 头结点的耗散值当作迭代深度 DEPTH，之后每次取超出 DEPTH 的最小耗散值作为迭代深度。其余执行部分与 A*搜索相同。

部分测试用的代码段注释掉了没有删除，助教有兴趣可以看看（雾。

问题二　X 数独问题

Part1 写在前面

使用的包有：

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
```

Windows 下直接编译运行即可。和问题一一样需要修改宏定义的 FILE_PATH 文件路径。

Part2 算法分析

数独问题没有特殊设计的数据结构，直接进行算法分析。

```
sudoku_recall(row, column, value):
if forward_check fail{
    recover domain image;
    return;
}// or do row, column, diagonal and domain check if not use forward_check
Use MRV to get next node; // or simply just choose next unfilled tile
if sudoku is done{
    Print and return;
}
for each value in node's domain{
    Sudoku_recall(new row, new column, new value);
}
recover domain image and present value; // here means assignment has conflict
return;
```

不过具体的实现和上述伪代码有少许不同。因为在每个块的值域中取值时出现了仍未发现问题的 bug，于是简单一点循环将 1-9 每个值尝试赋值，所以要多做冲突检测，但是前向检验仍然很好的跟踪到了潜在的冲突。

两份代码在主要结构上没有任何区别，只是将优化后的代码中使用了前向检验和 MRV 的代码段注释并替换成普通的寻找下一个节点的代码，就是普通的回溯搜索代码。

以样例二为例，普通的回溯搜索大约搜索了 13 万个节点，每进行一次回溯视为到达一个新结点，而使用了前向检验的代码大约搜索了 10 万个。两份代码在具体的执行时间上并没有出现太大的差异，前两个样例约几百毫秒。样例三时间长于一二，若测试请耐心（当然也没太长）。

值得注意的是，第三个样例，优化后的代码执行时间反而长些。认为应该是代码具体实现的问题，因为在代码中有大量的二重循环和赋值，维护前向检验和 MRV 的数据结构也花费了大量时间。代码优化的真正目的在于搜索树剪枝，在更大的问题中运行更细致编写的两份代码，结果肯定是高下立判的。

Part3 思考题

模拟退火算法和遗传算法可行，爬山算法不可行。

如模拟退火算法，先给数独一个随机的初始赋值。对原本是 0 的块做状态扩镇。如果数独的行/列/对角线/宫有重复数字，则升温，否则降温。目标状态的温度一定为 0，故可行。

但是使用爬山算法的话会因为被困在局部最优解而无法得出数独的解。

使用遗传算法的话，对未确定的数字进行随机赋值，用冲突数字最少做自然选择。在融合状态的时候不修改已确定的数字，也不对已确定的数字进行突变，即可。