AI实验一 "22数码问题"说明文档

PB15111662 李双利

- 实验说明
  - 文件结构

    在22Digits目录下共有四个cpp源代码文件、四个linux下的可执行文件以及该说明文档。

    输入文件为input.txt和target.txt，运行程序时需要将这两个文件放在该目录下。

    输出文件有四个分别为output_Ah1.txt、output_Ah2.txt、output_IDAh1.txt和output_IDAh2.txt。

  - 实验要求

    输入为一个5×5的矩阵，给定一个初始状态和目标状态，在棋盘上存在两个障碍物，只可左右穿过，不可上下穿过。通过搜索算法找到从初始状态移动到目标状态和最优解（移动路径）。

  - 运行环境

    OS：Ubuntu 16.04 lts 64位

    CPU：i5-7200U

    Memory：8GB

  - 编译执行说明

```
# 编译源代码
g++ Ah1.cpp -o Ah1
g++ Ah2.cpp -o Ah2
g++ IDAh1.cpp -o IDAh1
g++ IDAh2.cpp -o IDAh2
# 可以按照g++的-O3编译选项进行最高优化编译
# 在实验中用以下编译器优化的话时间可以减少50%~60%
# 为了获得最优的时间性能，在本实验目录下的可执行文件是采用-O3优化后的结果
g++ Ah1.cpp -O3 -o Ah1_O3
...
# 编译后执行
./Ah1
....
```

- 算法思想

  解决数码问题的核心思想是在每个位置选择四个方向（至多）进行下一深度的搜索，而这一过程的启发函数选择至关重要，选择的启发函数越接近真实代价效率越高。两种常见的启发函数分别为错位棋子数，和曼哈顿距离之和。对于本次实验，在常规的数码问题上增加了障碍，所以使得曼哈顿距离不可采纳，我采用的$h2$启发函数是在曼哈顿距离基础上针对障碍做了特殊的考虑。

此外，出于优化空间的考虑，对于棋盘我并没有采用5×5的二维的数组，而是采用的一维字符串数组。每个数字进行了加65的处理并转成char型，存储在数组中的状态中，字符'A'对应0，'B'~'w'对应1~22，'@'对应'-1'（即障碍）。

- 启发函数$h1$

  $h1$是错位的棋子数，代码如下：

```
int get_h1(string state)
{
    int wrong_num = 0;
    for(int i = 0;i < 25;i++)
        if(state[i] != target[i] && state[i] != 'A')
            wrong_num += 1;
    return wrong_num;
}
```

  $h1$相对简单，计算其值仅需要遍历状态数组，然后把错位的棋子数计数即可。在$h1$下，不在目标位置的数只需要一步的代价就归位，$h1$显然是可采纳的，并且这个估计值比实际代价会小很多，所以采用$h1$的搜索算法效果相对较差。

- 启发函数$h2$

  $h2$表示的是改进的曼哈顿距离，代码如下：

```
int get_h2(string state)
{
    int dis = 0, r1, c1, r2, c2;
    for(int i = 0;i < 25;i++)
        if(state[i] != '@' && state[i] != 'A')
        {
            int pos = target.find(state[i]);
            // 省略部分代码，此处计算得到两个位置(r1, c1)和(r2, c2)
            dis += abs(r1 - r2) + abs(c1- c2);
            // 是否左右穿越了障碍物
            if((r1 <= 2 && r2 >= 2) || (r1 >= 2 && r2 <= 2))
                dis -= column_check[c1][c2];
            // 是否上下会遇到障碍物
            if((c1 == 1 && c2 == 1) || (c1 == 3 && c2 == 3))
                dis += row_check[r1][r2];
        }
    return dis;
}
```

  对于$h2$启发函数，首先计算三个方向的曼哈顿距离，然后再特殊考虑障碍对于启发函数的影响，首先是左右穿过障碍的情况，如果直接按照原始的曼哈顿距离计算方法，一次穿越障碍的代价估计为2，但是实际代价仅为1，显然不可采纳，所以我对其做的修正如下：

  1. 如果一个棋子的当前位置到其目标位置路径可能会穿过障碍的话，减去1或者2（有可能是穿越了两个障碍），具体的代码实现是建立了`column_check()`，输入两个列位置就能$O(1)$时间做一个检测是否会穿越障碍(0次、1次或2次)的查询，这样求得的启发函数的估计代价必然是小于真实代价的，因为只要可能穿越障碍，那么就会按照穿越障碍的路径计算代价，$h2$必然是代价的一个下界。

2. 经过1步骤处理后的启发函数是可采纳的，但是并不是最接近于真实代价的启发函数，比如考虑这样一种情况：一个数在$(r1,c1)$，其目标位在$(r2,c1)$，e并且$c1=1\ or\ 3$（即障碍物所在的两列），$r1$和$r2$在障碍物所在行($r=2$)的两侧，这样从$(r1,c1)$到$(r2,c2)$时，其代价必然满足$c\geq|r1-r2|+2$，因为不能直接穿过障碍，所以最近也是要从无障碍的一侧绕过去，所以在计算启发函数$h2$时，检测这种情况，如果满足以上所述，对$h2$进行加2操作，具体代码实现是建立了`raw_check()`输入两个行位置就能$O(1)$时间做一个检测是否是上述情况的查询。这样的$h2$仍然是可采纳的，但是更接近于真实代价。

- A^*算法
  - a. 从初始状态 A 开始，将其作为一个待处理点加入一个开启列表中($open$)。并计算当前状态 A 的$f(n)=g(n)+h(n)$的值。(g 为路径耗散值，h 为启发函数值)。
  - b. 将开启列表中的初始状态取出，作为起点，并放入关闭列表($close$)。寻找起点周围所有的可达状态结点，如果已经在关闭列表中则跳过，否则也将他们加入开启列表。将到达该状态的行为记录下来，放到路径数组中。
  - c. 接着，从开启列表中寻找 f 值最小的状态结点，重复 c 步骤，直到找出的起点结点的h 值为 0。此状态即为目标状态。
  - d. 根据目标状态，从路径数组中依次读取动作，即得到从初始状态到达目标状态的动作序列。
- IDA^*算法

  IDA^*搜索与A^*搜索不同，它是一个深度优先遍历的过程，且不保留已经搜索过的状态。算法维护一个栈，程序一 开始将初始状态压入栈中。在一个循环中，若栈非空，弹出栈顶的状态，由这个状态扩展新的状态，并放入栈 中。直到找到目标状态，或者耗散值超过截断值，则结束这一结点的扩展。之后，选择超过截断值最小的耗散值，作为新的截断值，进入下一轮循环。

优化策略&复杂度分析

- A^*搜索

  为了优化算法，$open$集采用了优先队列来得到每一次耗散值最小的状态。每次仅需要$O(logn)$的代价维护优先队列。

  A^*搜索的时间复杂度为$O(b^d*logd)=O(b^d)$，其中b为平均每一个状态扩展后的状态数，d为搜索空间的最大深度。这里就是$O(4^d)$。为了减少搜索次数，我做了简单的剪枝操作，即每次搜索下一个状态时，会根据上一个动作作出剪枝，比如上一个动作是U，那么显然如果此次是D，就会回到之前的状态，属于重复搜索，所以每此都与上一次的动作作比较作出剪枝，每此扩展状态时最多加入三个子状态，所以时间复杂度变为$O(3^d)$ 。

  A^*搜索的空间复杂度较大。可以证明，除非启发函数与实际路径耗散的偏差的增长不超过实际路径耗散的对 数，即$|h(n)-h^*(n)|\leq O(logh^*(n))$，否则结点数量会呈现指数级的增长。

- IDA^*搜索

  与A^*相比，IDA^*最大的优点就是空间复杂度很小，大约为$O(d)$，其中d为最大搜索深度。 IDA^*的时间复杂度为$O(b^d)$，其中b为平均每一个状态扩展后的状态数，d为搜索空间的最大深度，采取剪枝后时间复杂度为$O(3^d)$ 。

结果分析

在我的PC环境上运行的结果如下（开启-O3优化）

样例参考步数/实际步数	A^*h1	A^*h2	IDA^*h1	IDA^*h2
5/5	$0s$	$0s$	$0s$	$0s$
10/10	$0s$	$0s$	$0s$	$0s$
15/15	$0s$	$0s$	$0s$	$0s$
20/20	$0s$	$0s$	$0s$	$0s$
25/25	$0.015625s$	$0s$	$0s$	$0s$
30/30	$0.0625s$	$0s$	$0.015625s$	$0s$
35/35	$0.9375s$	$0s$	$0.234375s$	$0s$
40/40	$32.8125s$	$0s$	$5.59375s$	$0s$
45/45	×	$0s$	$143.594s$	$0s$
50/50	×	$0.078125s$	$48min, 13s$	$0.015625s$
55/55	×	$2.76562s$	×	$0.625s$
60/60	×	$12.3125s$	×	$1.90625s$
70/70	×	×	×	$496.734s$
80/75	×	×	×	$1h, 32min, 18s$

1. 从实验结果来看，采用启发式函数$h2$的效果要远优于$h1$，在$steps$小于25时采用$h1$启发函数时运行时间几乎为0，而采用$h2$启发函数时在$steps$小于50时都接近于0，同等输入下，不管是A^*还是IDA^*搜索均是$h2$函数的效率更高。
2. 由于在我的环境下内存限制为8G（实际可用5G左右）。在$steps \geq 45$时，$Ah1$执行时会超出内存限制，无法执行。在$steps \geq 70$时，$Ah2$执行时会超出内存限制。
3. 再对比A^*算法和IDA^*算法，IDA^*除了空间性能显著优于A^*之外，在采用同一启发函数和相同输入时，IDA^*的时间性能也由于A^* 。