1951505 姜星宇

人工智能原理第 2 次作业

一、作业内容

A*搜索求解八数码问题.

二、实验原理

A*搜索是一种最佳优先搜索, 它的启发函数结合消耗与到终点的最短估计距离, 也可以认为是一致代价搜索对于排序依据的改进, 则同样需要用到一个优先队列与一个哈希表. 在八数码问题中, 消耗即为已经走过的步长, 而启发函数可以用所有方块分别到目标状态的步数之和来表示. 这样构造出的函数也满足A*算法的最优性条件.

八数码问题每一个排列都是一种状态, 那么共存在 9!=362880 种状态. 如果将这些状态看作结点的话, 那么这个数据结构就可以是一个存在 9!结点的无向图. 而连接结点的边则是相邻的状态, 每一个结点都有 2 到 4 条边. 而由于边的移动不改变逆序数的奇偶性, 整个图平均划分为两个独立的连通图. 八数码问题本质上就是在这个图的某一个连通子图上找到两个状态之间的(最短)路径.

同样的, 也有(n^2-1)数码问题. 除了偶数边长的问题会导致空白格的行数也对奇偶性产生影响外, 其他的与八数码问题基本一致, 只是结点数量迅速增加, A*算法很难在可接受的时间内找出最短路径.

三、实验内容

1. 源文件构成: (编译器 G++, MinGW-w64)

   每一个 source 文件夹中包含以下三个文件:

   - `game.h`: 处理程序内部逻辑的头文件.
   - `game.cpp`: 包含初始化, 搜索等算法与一些相关的宏与全局变量.
   - `main.cpp`: 处理窗体相关, 包含入口函数以及回调函数.

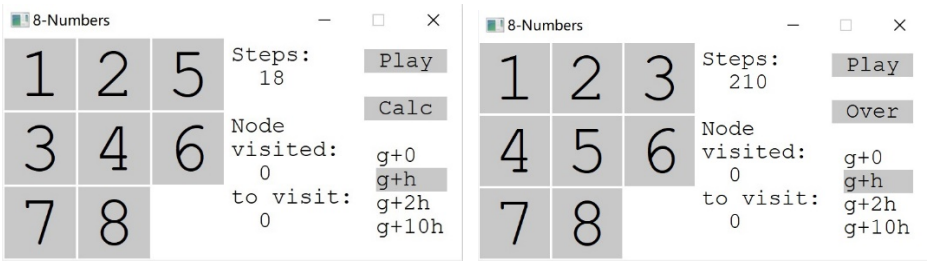
2. 主要数据结构:

   - `mat_struct` 结构体: 其中存储矩阵, 已走步数, 前驱结点以及启发函数值, 将大于号重载为启发函数值的比较. 在搜索时候用到.

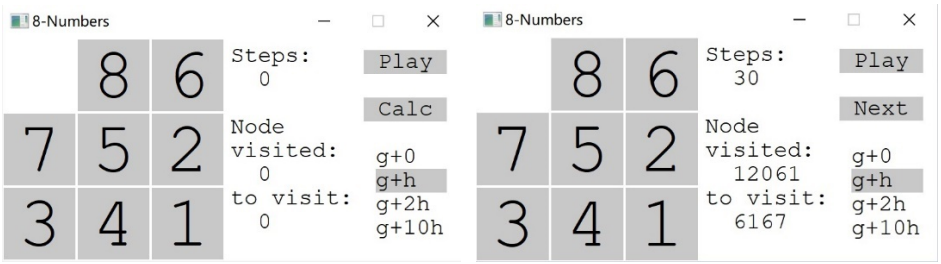
- `A_star_pq` 类型: 通过 STL 库中 `priority_queue` 模板类定义的关于 `mat_struct` 大于号重载后的最小优先队列, 实例化为变量 `pq`.
- `mat_tmp` 变量(`A_star()`内): 类型为 `vector<mat_struct>`, 用于存储已经访问过的结点以生成最终的路径.
- `seq` 变量(全局): 类型为 `stack<mat_struct>`, 用于先进后出地从后往前存储生成矩阵序列.

3. 主要函数:
   - `bool A_star()`: A*搜索主要函数. 负责创建优先队列并在 `while` 循环中进行结点扩展. 搜索到结果后将路径压入全局变量栈中.
   - `void initialize()`: 初始化过程, 负责将步长等参数初始化.
   - `void randomize()`: 随机生成满足条件的矩阵.
   - `int move()`: 改变全局变量 `mat`, 返回是否到达 `mat_fns` 目标状态.
4. 可视化实现方法: 调用 WinAPI, 创建实例进程句柄 `hInstance`, 并再次基础上建立 `hWnd` 窗体句柄, 用 `hdc` 设备内容句柄显示图形. 在回调函数 `WindowProc()`中处理鼠标点击, 刷新等信息.
5. 实验结果:
   - 八数码问题可视化界面及结果:

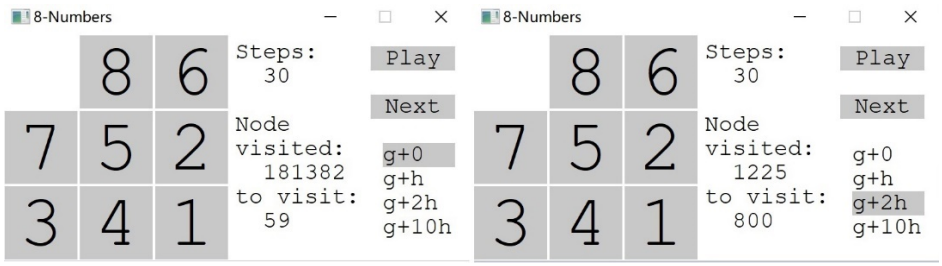
     鼠标点击移动方块达到最终状态并计数;



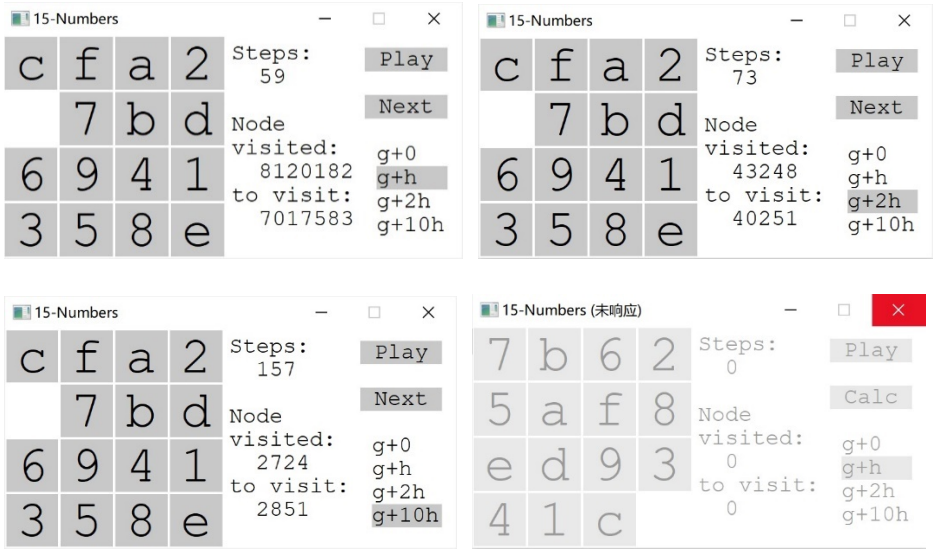
     点击 Play 重新生成并点击 Calc 计算选中启发函数的路径;



     选择不同的启发函数.



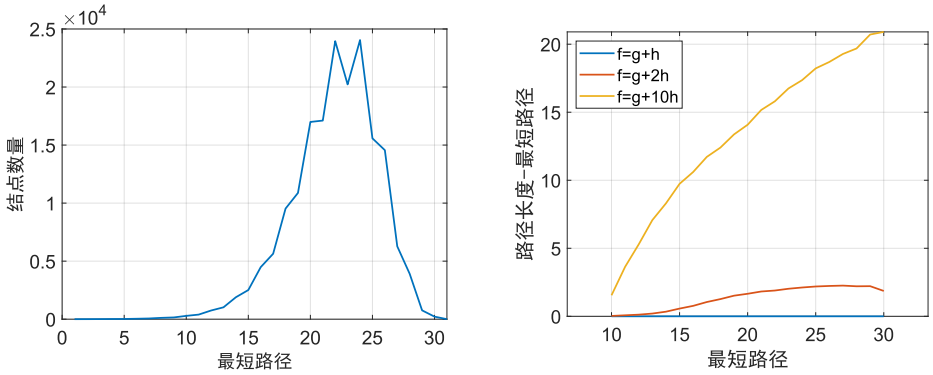
- 十五数码问题可视化界面及结果:



四、结果分析

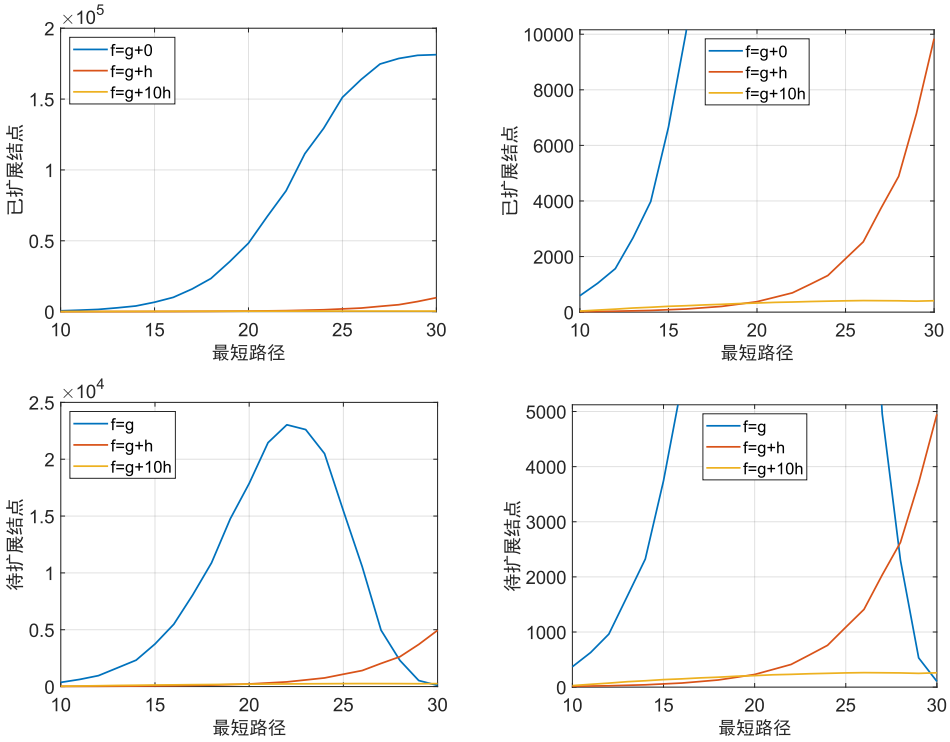
1. 八数码问题结果分析:

由于可解的八数码问题一共只有 9! / 2 = 181,440 种状态，而两个状态之间的路径由于对称性可以归结为这些状态到同一个标准状态的路径. 每一次搜索扩展的结点数量最多就是 181,440 种状态，而一次搜索的时间复杂度为 $O(n\log n)$, n 为进入优先队列中元素的个数. 因此将所有状态全部运行一遍是可行的，那么完全解出八数码问题的时间复杂度为 $O(n^2\log n)$, n 为结点数量. 因此经过十几个小时的运行后，得到以下结果.



上面第一张图是不同最短路径长度对应的节点个数. 可以看见搜索的步长大约处于 20 到 25 的范围内, 也就是说若采取一致代价或者深度优先搜索, 需要搜的层数大约是 22 层左右, 如果没有结点总数限制, 就是大约需要遍历 3^{22} 个结点, 也是不小的数字.

第二张图是不同启发函数计算出的路径长度比最短路径多出的步数. 若采取之前描述的 h 函数, 的确可以得到最优解. 如果将这个函数扩大两倍或是更多的倍数, 得到的结果往往就不是最优的解.



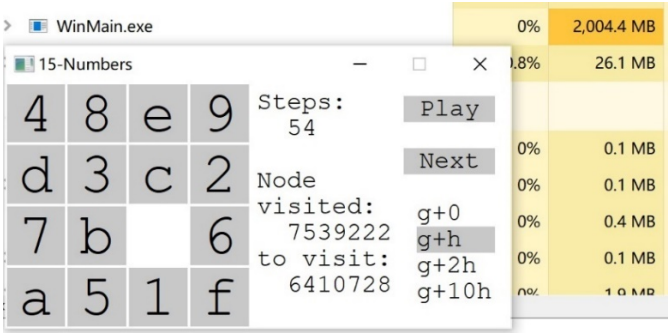
虽然得到的解不是最优的, 但是可以从随后的几张图看出, 当采取的搜索策略为一致代价搜索($h = 0$)(在这个问题中和宽搜是等价的, 从某种角度上来说也是一个经过堆优化的 Dijkstra 算法)时, 所扩展的结点比其他的启发函数要多得多, 当最短路径超过 25 后, 这个算法几乎将所有的结点遍历了一遍. 而对于 h 占主导作用的情形($h = 10h$), 扩展的结点相较于其他的情况并没有增加多少. 而满足最优解的A*算法大约就是这两者的平衡, 在保证最优解的情况下尽可能地减少结点扩展的数量, 这时的启发函数的确定就十分重要.

不过无论如何, 书上也说了, A*搜索在大部分情况下, 复杂度都是指数增长的, 对于上面的几个不同的启发函数只是分支的数量也就是指

数的底数有着区别，随着数据量的扩大，所需的时间都会指数级增加. 在八数码问题中由于结点数量的限制，这点体现的并不明显，如果扩展为 15 数码或者是更多的方块数量，就可以体现出这一点.

15 数码的结点数为 16! / 2 ≈ 10^{13}，这时也就是说没有了结点数的限制. 最后四张实验结果图中可以看见对于 59 步的路径，宽搜直接搜不出来. 而一致的启发函数已经扩展到了接近一千万个结点，耗时大约几十秒，增加启发函数的权重可以使时间控制在一秒内. 随着问题的规模继续增加，由于启发函数只是一个大概估计，不能根据启发函数一步走到底，很难满足多项式复杂度的条件，分支的结点数总归会大于 1，时间也会到达极限.

也像课本上所说，A*算法的局限性更多地体现在空间上，因为没有搜索完成时队列中所有的结点都不能释放，因此内存往往先超出限制. 如果资源能够做到用完随即释放，就算是几天等等也不是问题. 而下图大约几十秒运行时间的情况占用了大量的内存，当系统内存不够时，A*算法便无法计算出结果. (挺奇怪为什么算完还没有释放内存，查了一下发现是 STL 内存泄漏得厉害，要 swap 什么的处理，不过这次就不管了).



本来还想试试更多的方块，连 GUI 都写好了(source_n 文件夹内)，看到 15 数码都这样了，自动搜索路径就算了，手动玩玩好了.

2. 一些问题与思考:

在实际编程时遇到了一个问题：在某一些最短路径较长的情况下，利用这种方式构造出的 A*总是会计算出多两步的路径. 仔细比对后发现应当是因为没有在扩展结点时与队列中已扩展同样结点的 f 值进行比较并替换，导致这个正确的结点因为已经在队列中没有更新最短路径及前驱结点. 但由于直接用的 STL，因此没有办法确定某个结点在队列中的

地址，只好在这种情况下重新插入一遍. 不过我觉得如果启发函数是严格满足最优性条件的也就是 $h(n) < c(n, a, n') + h(n')$而不是小于等于，应当是可以不用判断的，这一点可以通过调整为浮点型并令 f 等于 $g + 0.999h$ 得以验证. 就这个例子证明的话大概就是因为不会在一个结点一直往深扩展(原先情况下 g 增加 1, h 减少 1, f 不变)，最多一层，这样一步并不会回到原来的结点，因此如果这个结点有更优的状态，一定是先被扩展进队列的.

3. 总结:

A*算法本质上是对于一致代价搜索的优化，并且可以附加一些条件证明出最优性的保持，我想这点应该是这个算法最重要的地方. 它在最短路径等问题上可以极大地减少扩展的结点数量. 也正是因为启发函数包含着人的思考方式，才更具有人工智能的特性.

五、文件清单

```
1951505-姜星宇-第二次作业
  |--实验报告.pdf
  |--8-Number (八数码问题)
  |   |--pic(结果图片)
  |   |   |--(…)
  |   |--source (源文件)
  |   |   |--game.cpp
  |   |   |--game.h
  |   |   |--main.cpp
  |   |--test (测试源文件)
  |   |   |--pic
  |   |   |   |--(结果图片)
  |   |   |--game.cpp
  |   |   |--game.h
  |   |   |--test.cpp
  |   |   |--output_whole.txt (所有可能情况的参数列表)
  |   |   |--test.m (处理参数并绘制图像的 MATLAB 代码)
  |   |--WinMain.exe (八数码问题可执行文件)
  |--15-Number ((n^2-1)数码问题)
  |   |--pic
  |   |   |--(结果图片文件)
  |   |--source_4 (15 数码源文件及可执行文件)
  |   |   |--(…)
  |   |--source_n ((n^2-1)数码源文件及可执行文件)
```