实验报告
PB13011079 杨智

实验环境:

ubuntu 14.04 LTS (x64)
memory : 7.7GB
CPU : Intel Core i7-4720HQ CPU @ 2.60GHz * 8
Hard Disk : 30.6GB

1. 立方数码问题

运行方式:

执行complie.sh之后，得到4个可执行文件，执行编译出来的可执行
文件即可。

注意事项:
1. 因为用的是相对路径，所以要在cpp文件的路径下编译和执行
2. 因为在A*算法中用了<unordered_set>来进行查重，所以编译选项要加上 —std=c++11，否则要把unordered_set 换成 set （之前发邮件问过助教，助教回复说可以用<unordered_set>的）
3. IDAh1.cpp和IDAh2.cpp中，有一行是 #define MAXNODE 999999999 ， 如果编译时报错或者运行时报错（segmentation fault）的话，可以把这个值改小一点，因为这个值是和机器相关的，在某些机器上这个值可能不合适。但是这个值改得太小的话可能会影响性能。

各个算法花费的时间（单位：毫秒**ms**）

（大于10分钟的记为F）

algo 步数/实际步数	Ah1	Ah2	IDAh1	IDAh2
5/5	0.085000	0.077000	0.080000	0.050000
10/10	0.120000	0.113000	0.093000	0.084000
15/15	0.118000	0.111000	0.029000	0.077000
20/16	0.705000	0.183000	0.365000	0.096000
25/25	234.697000	4.632000	75.754000	0.547000
30/30	1213.475000	0.460000	354.276000	0.352000
35/35	43590.756000	10.360000	5789.448000	4.209000
40/32	9809.191000	11.812000	1387.076000	7.875000
45/41	F	42.503000	599949.145000	15.780000
50/28	92.143000	1.277000	54.090000	0.407000
55/47	F	4544.653000	F	142.402000
60/50	F	4568.165000		195.786000
65/49	F	2314.216000		1399.361000
70/52	F	8666.733000		1045.622000
75/61	F	7831.728000		2575.595000
80/56	F	35396.611000		16193.676000
85/47	F	F		23387.919000
90/52	F	31.657000		15.473000
95/61	F			50448.681000
100/64	F			286414.341000

A* 算法伪代码:

// 已知h是可采纳的

queue为优先队列， 根据 f = g + h 来排序

```
queue.push(initial_state)
while (queue not empty)
        now_state = queue.pop
        if (now_state is goal_state)
                return found_solution
        queue.push(all the legal successor_states)
return found_no_solution
```

A*算法优化方法：

1. 用了unordered_set来进行查重，避免花费时间考虑重复的状态
2. 用27个char型来表示立方数码的状态，节省空间
3. 因为计算h2的代价比较大，所以实际上h2是根据搜索树上该结点的父结点的h2值来进行更新的，这样可以大大地减少计算量
4. 还有一些小的优化细节，不详述

A*算法的时间复杂度和空间复杂度：

A*算法的时间复杂度和heuristic的选择有关。当搜索空间没有限制的时候，展开的结点数是最短路径的长度的指数级，即O(b^d), 其中b是结点的平均后继数, d是最短路径的长度。

当满足以下三个条件的时候，A*算法的时间复杂度是最短路径长度的多项式：
1. 搜索空间是一棵树；
2. 只有一个目的结点；
3. ｜h(n) - h*(n)| <= O(log(h*(n)))， 其中h是真实的heuristic，h*是算法所选的heuristic函数。
否则，A*算法的时间复杂度是最短路径的长度的指数级。

通过分析可知道，A*(h1)的时间复杂度显然是O(b ^ d)，其中b是结点的平均后继数, d是最短路径的长度；A*(h2)的时间复杂度也是O(b ^ d)，这是根据实际测试得出来的，因为结点的平均后继数b为4、最短路径d的长度大于50时，近似地有 log(time) / log(b) / d = 0.13，其中time的单位为ms。

对于同样的heuristic函数，A*算法的时间复杂度和空间复杂度是一样的。

综上：
A*(h1)和A*(h2)的时间复杂度和空间复杂度都是O(b ^ d)，其中b是结点的平均后继数, d是最短路径的长度。

IDA * 算法伪代码：

```
node    current node
g       the cost to reach current node
f       estimated cost of the cheapest path (root..node..goal)
h(node)         estimated cost of the cheapest path (node..goal)
cost(node, succ)  step cost function
is_goal(node)     goal test
successors(node)  node expanding function

procedure ida_star(root)
        bound := h(root)
```

```
        loop
                t := search(root, 0, bound)
                if t = FOUND then return bound
                if t = ∞ then return NOT_FOUND
                bound := t
        end loop
end procedure

function search(node, g, bound)
        f := g + h(node)
        if f > bound then return f
        if is_goal(node) then return FOUND
        min := ∞
        for succ in successors(node) do
                t := search(succ, g + cost(node, succ), bound)
                if t = FOUND then return FOUND
                if t < min then min := t
        end for
        return min
end function
```

IDA*算法优化方法：

1. 用27个char型来表示立方数码的状态，节省空间
2. 因为计算h2的代价比较大，所以实际上h2是根据搜索树上该结点的父结点的h2值来进行更新的，这样可以大大地减少计算量
3. 因为运行的时候动态地进行malloc操作和free操作会花费很多时间，所以我编写了my_malloc和free_one函数。调用free_one的时候，只是把这个结点的内存空间的地址纪录在一个名为nodes[ ]的数组里面，而不会把这块内存空间交还给系统；当调用my_malloc的时候，会首先查看nodes[ ]是否为空，如果nodes[ ]不为空，则从nodes[ ]取一块已经分配了内存空间的地址，否则，调用malloc向系统请求一块新的空间。相当于人工地维护一个小型的堆栈，尽可能避免动态内存分配和释放。
4. 对于f值大于bound的结点，考虑完之后可以立即调用free_one函数进行释放；递归idastar_seach()函数的结点在该函数执行完之后也可以立即进行释放。这样一方面可以节省很多很多的空间，另一方面也可以和优化方法中的第三点配合在一起来节省时间。
5. 还有一些小的优化细节，不详述

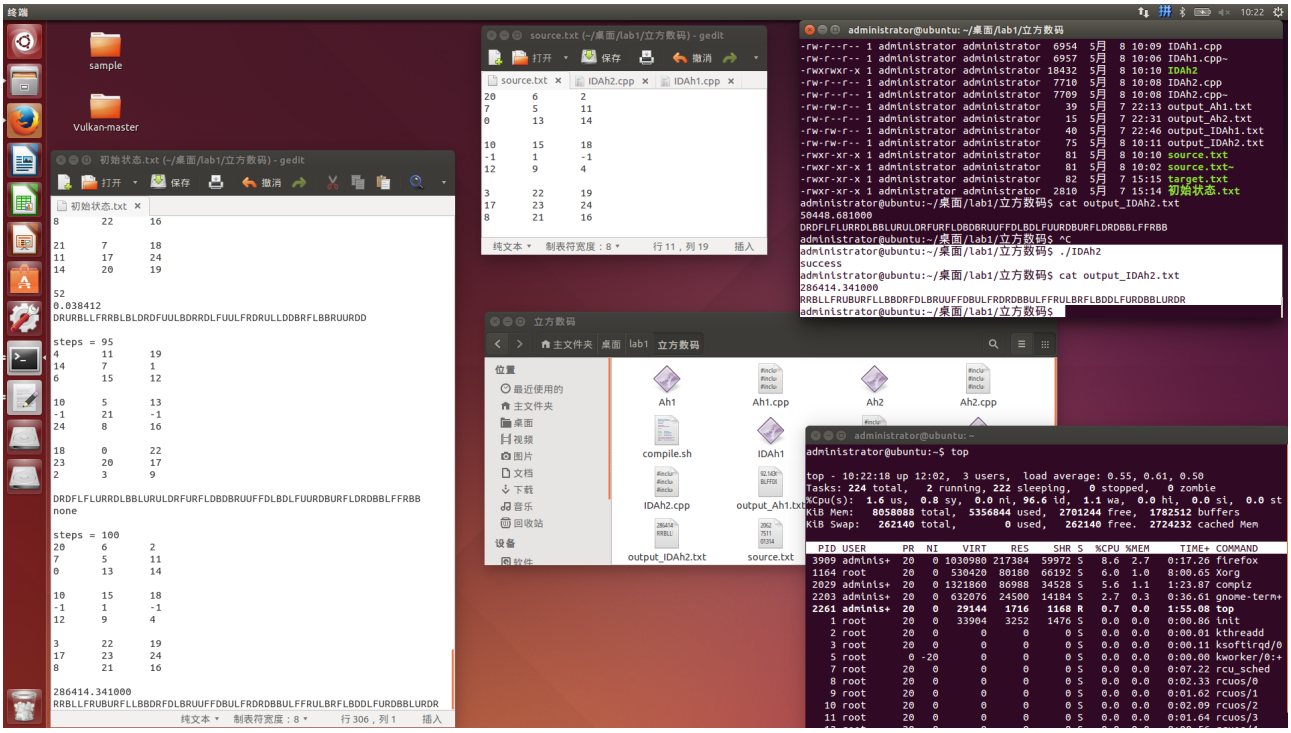
IDA*算法的时间复杂度和空间复杂度：

首先，根据2000年的一篇文章：Time complexity of iterative-deepening-A*（Richard E. Korf, Michael Reid, Stefan Edelkamp），在立方数码问题中，IDA*(h1)和IDA*(h2)的时间复杂度都是指数级的，即O(b ^ d)，其中b是结点的平均后继数, d是最短路径的长度。

但是，因为IDA*只需要记录当前路径上的结点（在优化方法3、4中可以看出来），所以IDA*的空间复杂度是O(d)，其中d是最短路径的长度。所以可以看出来IDA*(线性)比A*(指数级)要节省空间。

综上，IDA*(h1)和IDA*(h2)的时间复杂度都是O(b^d)，空间复杂度都是O(d)，其中b是结点的平均后继数, d是最短路径的长度。

实验结果：IDA*(h2)———Step_100



2. N皇后问题

运行说明

直接运行./compile.sh即可完成编译。
其中hill_climbing.cpp和simulated_annealing.cpp是对初始状态进行了优化的（效率较高），hill_climbing_silly.cpp和simulated_annealing_silly.cpp的初始状态是完全随机的（效率较低）。

说明：

只考虑了皇后不可以放在有障碍的地方，而没有考虑皇后放在障碍两边的情况。
可以证明：当n >= 4时，普通的n皇后问题必定有解，只有一个障碍、只考虑皇后不能放在有障碍的地方的n皇后问题也必定有解

hill_climbing算法和simulated_annealing算法花费的时间

（障碍物的位置为20 30）
（大于10分钟的记为F）

hill_climbing		simulated_annealing	
N的值	时间（ms）	N的值	时间（ms）
1000	2.562000	1000	18.184000
10000	38.507000	10000	85.151000
100000	338.773000	100000	1408.809000
(百万）1000000	7476.788000	(百万）1000000	21698.791000
3000000	31881.924000	3000000	91027.162000
5000000	61312.258000	5000000	
7000000	88907.357000	7000000	
9000000	107766.967000	9000000	

hill_climbing（N－queens）算法思想

以伪代码表述：

```
// board 是N皇后中的棋盘，以一维度数组表示
// 到达局部最优时重启，沿着最陡的方向爬山

hill_climbing(board)
{
        if （found_solution)
                return found
        while(1)
                // max 是冲突最多的皇后，min是这个皇后能到的冲突最少的位置
                max = max_conflict_queen
                min = min_conflict_place(max)
                if (min == board[max])
                        // 到了局部最优
                        break
                else
                        // 把记为max的queen移到记为min的位置
                        move_to(max, min)
}

hill_climbing_entry(board)
{
        initial(board)
        for (i = 0; i < RESTART_TIME; i++)
                hill_climbing(board)
                if (found_solution)
                        display(board)
```

```
                return 1
        else
                initial(board)
```

simulated_annealing（N-queens）算法思想

```
simulated_annealing(board)
{
        temperature = 0;
        delta = 0.005;
        while (1)
                // 随机选择一个皇后
                ii = random_queen()
                // cost 是皇后ii受到的冲突数
                cost  =  number_conflicts(ii)
                jj  =  random_move(ii)
                // next_cost 是把ii移到jj处后这个皇后受到的冲突数
                next_cost = number_conlicts(ii, jj)
                if ( try_this_move(cost, next_cost, temperature * delta))
                {
                        move_to(ii, jj)
                        if (found_solution)
                                return 1;
                temperature  = temperature + 1
}

try_this_move(cost, next_cost, parameter)
{
        // 当移动后会使代价变小的话，一定采取移动
        if (next_cost  < cost)
                return 1
        // 否则，以一个随迭代步数变化的概率决定是否采取移动
        if ( random(0, 1) <= exponent((cost - next_cost) * parameter) )
                return 1
        else
                return 0
}
```

QS4算法的算法思想（没有实现该算法）

QS4算法第一次在Rok Sosic、Jun Gu的paper “A polynomial time algorithm for the N-Queens problem”（1990） 中出现，基本思想是：

1. 选一个合适的常数M
2. 对n*n的n皇后问题，先在第 1 至 （n - M) 列完全没有冲突地放置 n － M个皇后
3. 最后M个皇后在 第 (n - M + 1) 至 n 列随机放置，只要满足没有行冲突和列冲突即可

4. 在第 (n - M + 1) 至 n 列用CSP算法求解

可以证明这个算法是多项式时间的。

（我没有用这个算法，因为感觉用这个算法有点不厚道，违背了这个实验本身的目的.....）

算法的空间问题（空间复杂度）

用一维数组来表示二维的棋盘状态来节省空间，因此此处空间复杂度是O(n)
在算法里面，用了3个vector<set<int>>来记录每行、每上对角线、每下对角线的皇后所在的位置，但平均下来也是O(n)
可以证明整个算法的空间复杂度是O(n)的。

另外， 因为本题中只有一个障碍，所以完全可以不用考虑得这么复杂，只要保证没有皇后处在障碍物上就行了，因此可以用一维数组来表示棋盘状态。

算法效率 （时间复杂度）

因为没有用QS4算法，所以显然时间复杂度是指数级的，O(a ^ n)。

主要从如下两个方面优化:
1. 尽可能选择冲突数少的初始状态，即在保证效率的情况下，初始化时尽可能地避免冲突
2. 算法内部的优化，主要优化底数，即O(a ^ n)中的a

a的值主要是靠实测得出的

在爬山算法中，
设time = [7.4768e+03 3.1882e+04 6.1312e+04 8.8907e+04 1.0777e+05]
设n = [1000000 3000000 5000000 7000000 9000000]
可得 log(time) ./ log(n) = [0.64562 0.69530 0.71467 0.72299 0.72366], 可见的确是近似地为O(a ^ n)的

在模拟退火算法中，
设time = [8.5151e+01 1.4088e+03 2.1699e+04 9.1027e+04]
设n = [10000 100000 1000000 3000000]
可得 log(time) ./ log(n) = [0.48255 0.62977 0.72274 0.76564]，可见n足够大时是渐进O(a ^ n)的

实验结果说明

实现结果如前面列出的时间表所示，
用冲突较少的初始状态的话（初始化后还是指数时间的搜索，而非用QS4算法或者直接使用n皇后问题的数学解），在两分钟之内，爬山算法可以解决千万级别的n皇后问题， 模拟退火算法可以解决三百万个皇后的问题