全局搜索算法的实验分析

实验零：补全代码并填写到实验报告中（报告中只需填写 C++实现）

breadth_first_search.hpp

```
    if (tree_search){   // tree 不用判重
                    // 扩展待访问的结点集合——1 行
                    states_queue.push(new_state);
                    if (require_path){
                        // 在 last_state_of 中记录路径——1 行
                        last_state_of[new_state] = state;
                    }
                }    else    if    (explored_states.find(new_state)    ==
    explored_states.end()){

                    // 扩展待访问的结点集合——2 行
                    explored_states.insert(new_state);
                    states_queue.push(new_state);

                    if (require_path){
                        // 在 last_state_of 中记录路径——1 行
                        last_state_of[new_state] = state;
                    }
                }
```

depth_first_search.hpp

```
    // 如果当前状态还有动作未尝试过
            if (action_id < state.action_space().size()){

                // 当前状态待尝试的动作变为下一个动作，等待回溯的时候尝试——
—1 行
                states_stack.push(std::make_pair(state, action_id + 1));

                // 尝试当前动作，获得下一步状态
                new_state = state.next(state.action_space()[action_id]);

                if (tree_search){
                    // 扩展待访问的结点集合——1 行
                    states_stack.push(std::make_pair(new_state, 0));
                    if (require_path){
                        // 记录路径——1 行
                        last_state_of[new_state] = state;
                    }
                }    else    if    (explored_states.find(new_state)    ==
```

```
explored_states.end()){

                    // 扩展待访问的结点集合——2 行
                    explored_states.insert(new_state);
                    states_stack.push(std::make_pair(new_state, 0));

                    if (require_path){
                        // 记录路径——1 行
                        last_state_of[new_state] = state;
                    }
                }
            }
```

heuristic_search.hpp

```
    // 从开结点集中估值最高的状态出发尝试所有动作
            for (ActionType action : state.action_space()){

                // 状态转移——1 行
                new_state = state.next(action);

                // 如果从当前结点出发到达新结点所获得的估值高于新结点原有的
估值，则更新
                if (best_value_of.find(new_state) == best_value_of.end()
                    or value_of(new_state) > best_value_of[new_state]){
                    // 更新状态价值，扩展待访问的节点集——2 行
                    best_value_of[new_state] = value_of(new_state);
                    states_queue.push(new_state);
                    // 记录路径——1 行
                    last_state_of[new_state] = state;
                }
            }
```

实验一：程序语言和 I/O 对程序运行时间的影响

实验目的：了解 C++和 Python 语言程序的运行效率及 I/O 在程序运行时间中的占比

实验步骤：分别运行 2 次 C++和 Python 版本的宽度优先搜索解 11 和 12 皇后问题的样例程序，一次在显示器上输出解，一次注释掉输出解的语句，比较它们找到全部解的运行时间。全部采用树搜索。

实验结果：

时间（秒）	C++程序		Python 程序	
	输出解	不输出解	输出解	不输出解
11	1	0	6.348	4.22
12	9	1	33.19	22.58

实验结果分析：

1. 解决 N 皇后问题时，C++程序执行效率显著高于 Python 程序，C++在大规模计算和搜索的场景下执行效率优势显著。
2. I/O 操作导致两种语言的运行时间均显著增加，在计算中可以避免不必要的输出，以提高程序运行效率。

实验二：深度优先和宽度优先的时间效率比较

实验目的：比较深度优先和宽度优先算法在求解 N 皇后问题上的时间效率

实验步骤：分别运行 C++和 Python 版本的宽度优先和深度优先搜索算法求解 N 皇后问题，注释掉输出解的语句，比较它们找到全部解的运行时间。全部采用树搜索

实验结果：

时间（秒）	宽度优先		深度优先	
	C++	Python	C++	Python
8 皇后	0	0.044	0	0.048
9 皇后	0	0.194	0	0.204
10 皇后	0	0.874	0	0.887
11 皇后	0	4.22	0	4.41
12 皇后	1	22.61	2	22.75
13 皇后	6	242.48	13	138.40
14 皇后	35	>300	79	>300
15 皇后	>300	>300	>300	>300
……				

实验结果分析：

1. 在求解 N 皇后问题时，对于较小的 N 值（8-11），BFS 和 DFS 在 C++和 Python 中运行时间都较短且差异不是很大，说明对于小规模的问题，两种搜索策略效率相近，编程语言对运行时间绝对值的影响也不大。
2. 随着 N 值的增加，BFS 和 DFS 的求解时间均对 N 呈现指数形增加，且 Python 运行时间显著长于 C++，更早开始出现运行超时的情况。
3. 对于同一种语言，N 值对运行时间的影响比搜索方法（DFS/BFS）对运行时间的影响更显著。

实验三：深度优先和宽度优先的空间效率比较

实验目的：比较深度优先和宽度优先算法在求解 N 皇后问题上的空间效率

实验步骤：分别运行 C++和 Python 版本的宽度优先和深度优先搜索算法求解 N 皇后问题，注释掉输出解的语句，比较它们找到全部解时开节点集同时存储的最大节点数。全部采用树搜索

实验结果：

空间效率（节点数）	宽度优先		深度优先	
	C++	Python	C++	Python
8 皇后	573	573	9	9
9 皇后	2295	2295	10	10
10 皇后	9643	9643	11	11
11 皇后	44235	44235	12	12
12 皇后	223174	223174	13	13
13 皇后	1161451	1161451	14	14

14 皇后	6573621	-	15	-
15 皇后	-	-	-	-
……				

实验结果分析：

1. 空间开销虽 N 值增大而增大，且 BFS 中开销随 N 呈现指数级增长，DFS 呈现线性增长。
2. 对相同 N 值，C++和 Python 的 BFS 空间开销远大于 DFS 空间开销，这说明 DFS 在空间效率上显著优于 BFS，在内存有限时可以解决更大规模的问题。
3. 使用 C++和 Python 的相同算法对空间效率无显著影响，空间开销主要取决于算法本身和问题规模。

实验四：比较不同算法求解最短路径问题

实验目的：比较分析一致代价、贪心、A*算法求解最短路径问题的差异

实验步骤：在根据要求完善代码的基础上，分别运行一致代价、贪心、A*算法求解最短路径问题的程序（C++或者 Python，语言不限），填写并分析实验结果。

实验结果（使用 Python 实现）：

	进入优先队列顺序	输出的解路径	解路径的花费
一致代价	2 3 6 8 16	2 3 5 1 0 7 4 6 8 11 9 16 10 16	418
贪心搜索	2 3 4 16	2 3 5 1 4 6 0 16	450
A*	2 3 6 8 16	2 3 5 1 4 6 0 8 11 16 16	418

实验结果分析：

1. 解的完备性：
   一致代价搜索和 A * 搜索：保证找到最短路径。二者是完备（A*算法在启发式函数是一致的时候是完备的）。
   贪心搜索：路径花费 450，局部最优而非全局最优。是不完备的。
2. 执行效率：
   一致代价搜索：扩展节点多、效率低，适用于必须全局最优但无启发式引导的场景。
   贪心搜索：扩展节点少、速度快，但不完备，适用于只需找到较优解的场景。
   A * 搜索：完备且最优，效率高于一致代价搜索，但需要一致的启发式函数。

1. 截图【一致代价搜索：进优先队列顺序、输出的解路径、解路径的花费】
2. 截图【贪心搜索：进优先队列顺序、输出的解路径、解路径的花费】
3. 截图【A*搜索：进优先队列顺序、输出的解路径、解路径的花费】

```
一致代价
<begin>
At: 2, from edge None, distance: 0
At: 3, from edge 8, distance: 140
At: 6, from edge 22, distance: 220
At: 8, from edge 28, distance: 317
At: 16, from edge 30, distance: 418
<end>
Order of entry into priority queue:
2 3 5 1 0 7 4 6 8 11 9 16 10 16

贪心
<begin>
At: 2, from edge None, distance: 0
At: 3, from edge 8, distance: 140
At: 4, from edge 24, distance: 239
At: 16, from edge 26, distance: 450
<end>
Order of entry into priority queue:
2 3 5 1 4 6 0 16

A*
<begin>
At: 2, from edge None, distance: 0
At: 3, from edge 8, distance: 140
At: 6, from edge 22, distance: 220
At: 8, from edge 28, distance: 317
At: 16, from edge 30, distance: 418
<end>
Order of entry into priority queue:
2 3 5 1 4 6 0 8 11 16 16
```