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

```
// 考虑所有可能动作，扩展全部子状态
for (ActionType action : state.action_space()) {
    new_state = state.next(action);
    if (tree_search) {
        states_queue.push(new_state);
        if (require_path) {
            last_state_of[new_state] = state;
        }
    }
    else if (explored_states.find(new_state) ==
explored_states.end()) {
        states_queue.push(new_state);
        explored_states.insert(new_state);
        if (require_path) {
            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()) {
    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) {
        states_stack.push(std::make_pair(new_state, 0));
        if (require_path) {
            last_state_of[new_state] = state;
        }
    }
    else if (explored_states.find(new_state) ==
explored_states.end()) {
        states_stack.push(std::make_pair(new_state, 0));
        explored_states.insert(new_state);
        if (require_path) {
            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()) {

    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]) {
        states_queue.push(new_state);
        best_value_of(new_state) = value_of(new_state);
        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	4.68	4.13
12	3	1	26.49	24.23

实验结果分析：

1. 在解决 N 皇后问题时，C++ 程序的效率显著高于 Python 程序，这说明了 C++在执行效率上明显优于 Python，尤其是在需要大量计算和搜索的场景中。
2. I/O 操作会增加程序的运行时间，无论是 C++ 还是 Python。因此，在高效计算任务中，应尽量减少不必要的 I/O 操作。

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	0.045	0	0.047
9 皇后	0	0.189	0	0.198
10 皇后	0	0.838	0	0.868

11 皇后	0	4.184	0	4.246
12 皇后	0	23.831	0	22.776
13 皇后	2	139.348	2	129.174
14 皇后	10	>300	10	>300
15 皇后	62	>300	65	>300
……				

实验结果分析：

1. 随着问题规模增大，无论是 Python 还是 C++，无论是 BFS 还是 DFS，求解时间均显著增加，且与问题规模 N 大致呈指数关系。
2. C++ 实现的算法在大规模问题上明显优于 Python 实现的算法。对于 14 皇后以上的问题规模，Python 已无法在给定时限内求解，而 C++ 仍可以正常求解。
3. 对于同一种语言的实现方式，BFS/DFS 的所需时间相近。

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	572	9	8
9 皇后	2295	2294	10	9
10 皇后	9643	9642	11	10
11 皇后	44235	44234	12	11
12 皇后	223174	223173	13	12
13 皇后	1161451	1161450	14	13
14 皇后	6573621	/	15	/
15 皇后	39933409	/	16	/
……				

实验结果分析：

1. 随着问题规模增加，BFS/DFS 求解所需空间均增加。
2. BFS 所需空间同 N 大致呈指数增加关系。
3. DFS 所需空间同 N 大致呈线性增加关系。

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：

	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 5 1 0 7 4 6 8 11 9 16 10 16	2 3 6 8 16	418

贪心搜索	2 3 5 1 4 6 0 16	2 3 4 16	450
A*	2 3 5 1 4 6 0 8 11 16 16	2 3 6 8 16	418

实验结果分析：

1. 一致代价搜索是完备的，其估值函数 $f(n)=g(n)+0$，其中 $g(n)$ 是路径代价函数。只要路径代价是有限的，并且每一步的代价都是正数，它能够找到最优解（完备+最优）。它的时间、空间复杂度均为$O\left(b^{1+\lfloor C^*/\epsilon \rfloor}\right)$。
2. 贪心搜索是不完备的，它可能陷入局部最优解而无法找到全局最优解，尤其在启发式函数不完美的情况下。它的时间、空间复杂度均为 $O(b^m)$，其中 b 是分支因子，m 是最大深度。
3. A* 搜索是条件完备的。对于图搜索来说，如果估值函数具有一致性，那么 A* 搜索也是最优的。
4. 就结果来看，空间占用 UCS>A*>贪心；解的正确性：A*、UCS 得到了最优解，而贪心陷入了局部最优解。

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】

```
arthals in  Aurora in Homework/05-Global-Search/python via  v3.11.8 via  torch  took 39ms
at 19:26:38 ❯ python short_path_ucs.py
<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
<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
<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
```