**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){

                    // TODO：扩展待访问的结点集合——1行

                    states\_queue.push(new\_state);

                    if (require\_path){

                        // TODO：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()){

                    // TODO：扩展待访问的结点集合——2行

                    states\_queue.push(new\_state);

                    explored\_states.insert(new\_state);

                    if (require\_path){

                        // TODO：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

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

states\_stack.push(std::make\_pair(new\_state, 0));

if (require\_path){

// 记录路径

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

states\_stack.push(std::make\_pair(new\_state, 0));

explored\_states.insert(new\_state);

if (require\_path){

// 记录路径

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()){

                // TODO：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]){

                    // TODO：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);

                    // TODO：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 | 2 | 0 | 5.615999460220337 | 3.988999366760254 |
| 12 | 12 | 1 | 30.820499420166016 | 21.660499811172485 |

实验结果分析：

可以看到在任何情况下都是c++比python要快很多，都是快了一倍左右，尤其是不输出解的情况下运行12皇后，c++直接时间是py的1/20。横向对比输出与否，可以看到c++中输出结果耗费的时间是远大于运算本身的，程序的绝大部分时间都用在打印结果上了。而python中打印结果的时间基本上是整个运行时间的三分之一，大部分时间还是用在运算上，打印与否的时间差异就是三分之一左右，不算太大。

**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 | 0.04049992561340332 | 0 | 0.04399991035461426 |
| 9皇后 | 0 | 0.18790745735168457 | 0 | 0.20550036430358887 |
| 10皇后 | 0 | 0.8289995193481445 | 0 | 0.8600003719329834 |
| 11皇后 | 0 | 3.9760005474090576 | 1 | 4.0140016078948975 |
| 12皇后 | 1 | 22.036499977111816 | 2 | 21.810999393463135 |
| 13皇后 | 5 | 124.90235471725464 | 13 | 122.06918001174927 |
| 14皇后 | 30 | >300 | 68 | >300 |
| 15皇后 | 301 | >300 | 555 | >300 |
| …… |  |  |  |  |

实验结果分析：不同语言之间，仍然是c++效率远比python快。对于c++，bfs基本上时间是dfs的一半，而python的话两者相差不大。

**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 | 9463 | 9463 | 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 | 39933409 | / | 16 | / |
| …… |  |  |  |  |

实验结果分析：显然不同语言用的空间节点数是一样的（实现了相同逻辑的搜索）。可以看到，bfs用了比dfs多得多的节点储存数，尤其是当N的规模增大时，由于状态数指数级增长，bfs用的空间开销也会指数级增长。然而dfs的栈储存的节点数等于深度N再加1（满时），所以其空间开销是很少的。

**实验四：比较不同算法求解最短路径问题**

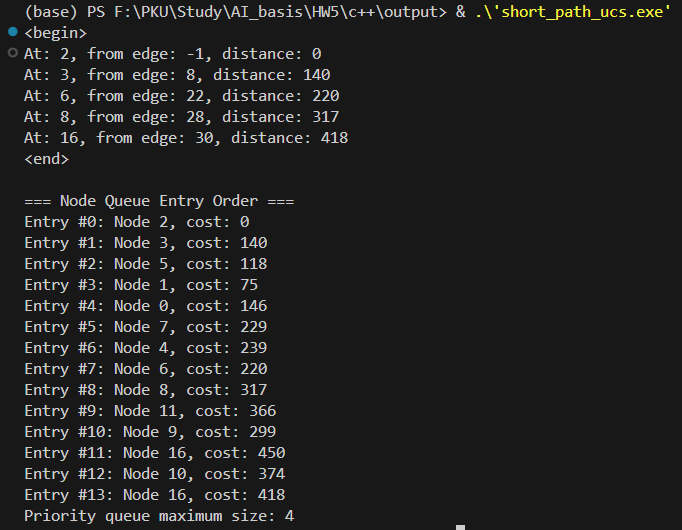
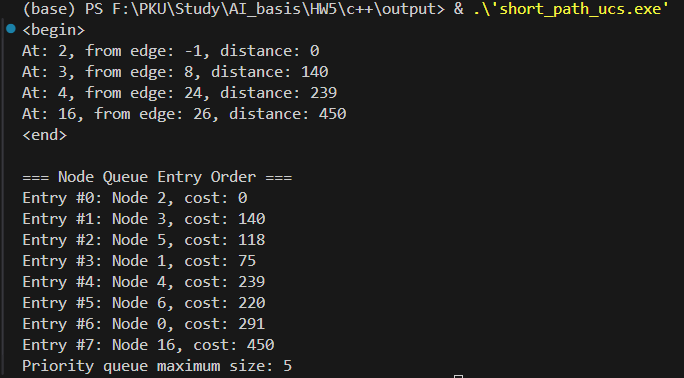
实验目的：比较分析一致代价、贪心、A\*算法求解最短路径问题的差异

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 | 2->3->6->8->16 | 418 |

实验结果分析：可以看到一致代价搜索和A\*都给出了正确的结果，而贪心搜索并没有找到最优解，但是它展开的节点最少。

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】

