**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

// 扩展待访问的结点集合：

if ( action\_id < state.action\_space().size() ) {

states\_stack.push( std::make\_pair( state, action\_id + 1 ) );

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.insert( std::make\_pair( state, new\_state ) );

}

}

else if ( explored\_states.find( new\_state ) == explored\_states.end() ) {

explored\_states.insert( new\_state );

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

if ( require\_path ) {

last\_state\_of.insert( std::make\_pair( state, new\_state ) );

}

}

}

// ……

depth\_first\_search.hpp

if ( action\_id < state.action\_space().size() ) {

states\_stack.push( std::make\_pair( state, action\_id + 1 ) );

new\_state = state.next( state.action\_space()[ action\_id ] );

if ( tree\_search ) {

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

}

else if ( explored\_states.find( new\_state ) == explored\_states.end() ) {

explored\_states.insert( new\_state );

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

}

}

// ……

heuristic\_search.hpp

//…...

for ( ActionType action : state.action\_space() ) {

new\_state = state.next( action );

if ( best\_value\_of.find( new\_state ) == best\_value\_of.end() or value\_of( new\_state ) > best\_value\_of[ new\_state ] ) {

best\_value\_of[ new\_state ] = value\_of( new\_state );

states\_queue.push( new\_state );

last\_state\_of.insert( std::make\_pair( state, new\_state ) );

}

}

}

// ……

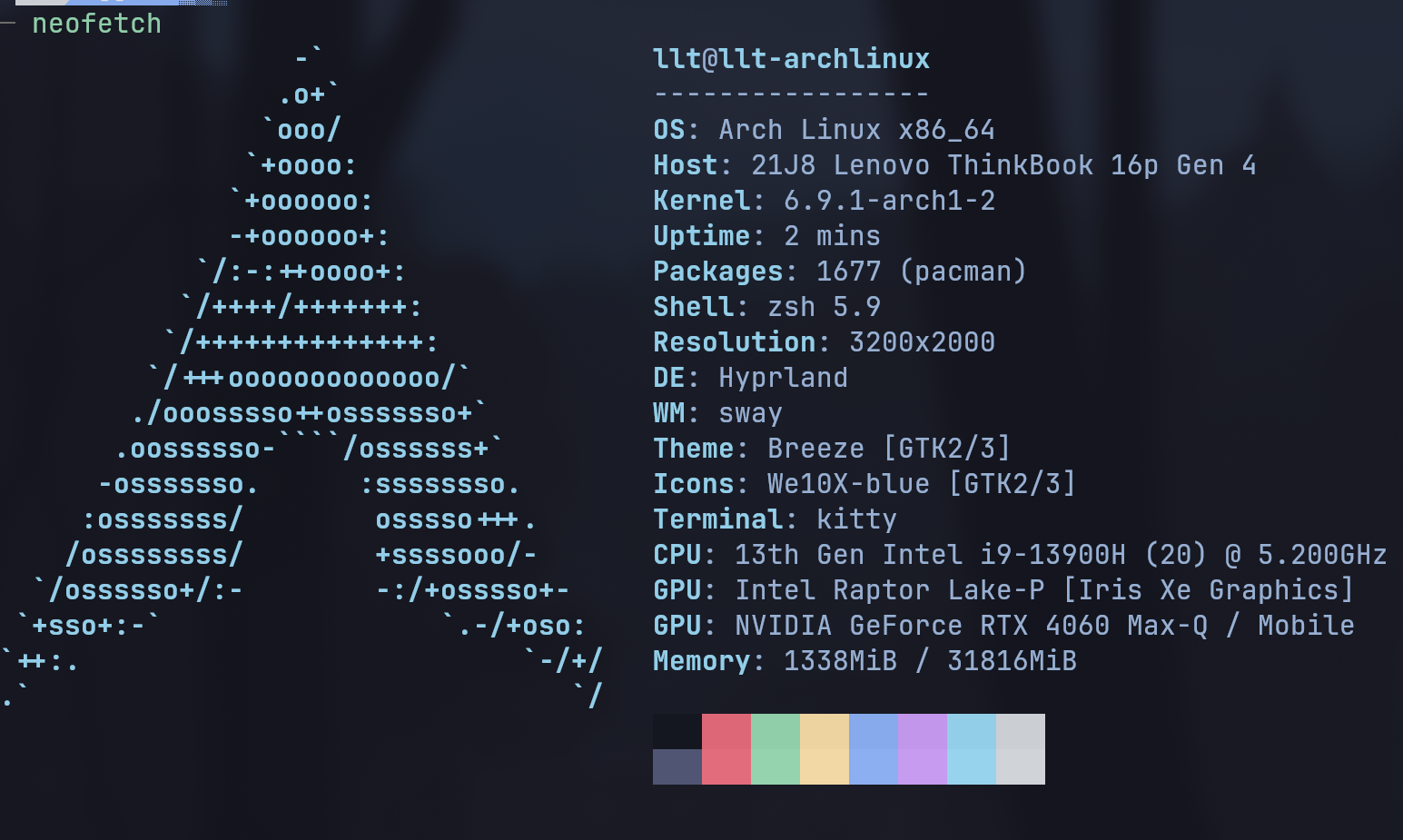
**实验一：程序语言和I/O对程序运行时间的影响**

测试环境（下皆同）：

g++ 14.1.1

python 3.11.7

编译参数：-lm -std=c++20 -Wall -Wextra -Wshadow -Dllt



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 | 0.278s | 0.233s | 3.529s | 3.396s |
| 12 | 1.478s | 1.235s | 19.810s | 19.157s |

实验结果分析：

从输出解和不输出解来看，输出解所用的时间略大于不输出解，可以看出IO确实对程序运行效率有影响，但影响有限，可能是IO量不够大的原因。分语言来看：cpp语言在11皇后问题上IO用时为0.045s,12皇后问题IO用时为0.24s;python在11皇后上IO用时为0.133s,在12皇后上输出解用时为0.633s,显著大于c++语言，可以看出c++语言运行及IO效率都远高于python。

**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 | 0.005s | 0.036s | | 0.008s | 0.037s |
| 9皇后 | 0.019s | 0.151s | | 0.031s | 0.163s |
| 10皇后 | 0.054s | 0.695s | | 0.105s | 0.708s |
| 11皇后 | 0.233s | 3.460s | | 0.482s | 3.470s |
| 12皇后 | 1.240s | 19.034s | | 2.466s | 18.326s |
| 13皇后 | 6.878s | 111.824s | | 13.523s | 105.145s |
| 14皇后 | 40.175s | >300s | | 79.13s | >300s |
| 15皇后 | 248.53s | >300s | | 811s | >300s |
| …… |  |  | |  |  |

实验结果分析：

从c++来看，宽度优先搜索算法运行时间是深度优先搜索的一半左右， 而从python版本来看两者时间差异不大。根据背景知识，两者的时间复杂度应当一致，对于cpp版本的常数差异，个人认为首先是二者理论上确实有一个常数级的复杂度差异，其次怀疑python对于堆栈的实现较优或对于队列实现较劣，从而产生了常数时间复杂度差异。总而言之，宽搜和深搜二者时间接近。

对于时间复杂度，可以观察到二者耗时对于问题规模都非常敏感，时间消耗关于问题规模增长很剧烈。所以，使用全局搜索可能不适合解决大规模的问题，需要使用其他算法。

**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 | >300s | | 15 | >300s |
| 15皇后 | 39933409 | >300s | | 16 | >300s |
| …… |  |  | |  |  |

实验结果分析：

先分语言讨论：可见c++和python内存消耗接近，差了1应当是统计方法不同造成的。

在分方法：显然，宽度优先搜索对内存的消耗显著大于深度优先搜索，猜想广搜在后期速度下降可能也有内存超限需要调取虚拟内存的原因。

**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 | 3 5 1 0 7 4 6 8 11 9 16 10 16 | 2->3->4->16 | 418 |
| 贪心搜索 | 3 5 1 4 6 0 16 | 2->3->4->16 | 450 |
| A\* | 3 5 1 4 6 0 8 11 16 16 | 2->3->4->16 | 418 |

实验结果分析：

贪心法进队数量最小，但最短路求错了。A\*数量小的同时，最短路也是正确的，一致代价搜索最多，解也对。总而言之，正确实现的A\*效率高，效果好，是个很合适的选择。

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】

