

# P01 Pacman Game

---

18364066 Yanzuo Lu

September 26, 2020

## Contents

<b>1</b>	<b>Search in Pacman</b>	<b>3</b>
1.1	Introduction . . . . .	3
1.2	Welcome to Pacman . . . . .	4
1.3	Question 1: A* search ( 3 points) . . . . .	4
1.4	Question 2: Corners Problem: Heuristic ( 3 points) . . . . .	4
1.5	Question 3: Eating All The Dots ( 4 points) . . . . .	6
<b>2</b>	<b>Multi-Agent Pacman</b>	<b>7</b>
2.1	Introduction . . . . .	7
2.2	Multi-Agent Pacman . . . . .	8
2.3	Question 4: Minimax ( 5 points) . . . . .	8
2.4	Question 5: $\alpha - \beta$ Pruning ( 5 points) . . . . .	9
<b>3</b>	<b>Notes</b>	<b>10</b>
<b>4</b>	<b>Results</b>	<b>11</b>
4.1	Question 1: A* search . . . . .	11
4.2	Question 2: Corners Problem: Heuristic . . . . .	11

4.3	Question 3: Eating All The Dots . . . . .	12
4.4	Question 4: Minimax . . . . .	13
4.5	Question 5: $\alpha - \beta$ Pruning . . . . .	14
4.6	Summary . . . . .	15

# 1 Search in Pacman

## 1.1 Introduction

In this section, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios.

---

**Files you'll edit:**

<code>search.py</code>	Where all of your search algorithms will reside.
<code>searchAgents.py</code>	Where all of your search-based agents will reside.

---

**Files you might want to look at:**

<code>pacman.py</code>	The main file that runs Pacman games. This file describes a Pacman GameState type.
<code>game.py</code>	The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
<code>util.py</code>	Useful data structures for implementing search algorithms.

---

**Supporting files you can ignore:**

<code>graphicsDisplay.py</code>	Graphics for Pacman
<code>graphicsUtils.py</code>	Support for Pacman graphics
<code>textDisplay.py</code>	ASCII graphics for Pacman
<code>ghostAgents.py</code>	Agents to control ghosts
<code>keyboardAgents.py</code>	Keyboard interfaces to control Pacman
<code>layout.py</code>	Code for reading layout files and storing their contents
<code>autograder.py</code>	Project autograder
<code>testParser.py</code>	Parses autograder test and solution files
<code>testClasses.py</code>	General autograding test classes
<code>test_cases/</code>	Directory containing the test cases for each question
<code>searchTestClasses.py</code>	Specific autograding test classes

---

**Files to Edit and Submit:** You will fill in portions of `search.py` and `searchAgents.py` during the assignment. You should submit these files with your code and comments. Please do not change the other files in this distribution or submit any of our original files other than these files.

**Evaluation:** Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder.

## 1.2 Welcome to Pacman

After downloading the code (search.zip), unzipping it, and changing to the directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

The simplest agent in `searchAgents.py` is called the `GoWestAgent`, which always goes West (a trivial reflex agent). This agent can occasionally win:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

But, things get ugly for this agent when turning is required:

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

If Pacman gets stuck, you can exit the game by typing `CTRL-c` into your terminal.

Soon, your agent will solve not only `tinyMaze`, but any maze you want.

Note that `pacman.py` supports a number of options that can each be expressed in a long way (e.g., `--layout`) or a short way (e.g., `-l`). You can see the list of all options and their default values via:

```
python pacman.py -h
```

Also, all of the commands that appear in this project also appear in `commands.txt`, for easy copying and pasting.

## 1.3 Question 1: A\* search ( 3 points)

Implement A\* graph search in the empty function `aStarSearch` in `search.py`. A\* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The `nullHeuristic` heuristic function in `search.py` is a trivial example.

You can test your A\* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as `manhattanHeuristic` in `searchAgents.py`).

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```

## 1.4 Question 2: Corners Problem: Heuristic ( 3 points)

*Note: Make sure to complete Question 1 before working on Question 2, because Question 2 builds upon your answer for Question 1.*

Implement a non-trivial, consistent heuristic for the `CornersProblem` in `cornersHeuristic`.

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

*Note:* `AStarCornersAgent` is a shortcut for

-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic.

**Admissibility vs. Consistency:** Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be admissible, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be consistent, it must additionally hold that if an action has cost  $c$ , then taking that action can only cause a drop in heuristic of at most  $c$ .

Remember that admissibility isn't enough to guarantee correctness in graph search – you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in  $f$ -value. Moreover, if UCS and A\* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

**Non-Trivial Heuristics:** The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

**Grading:** Your heuristic must be a non-trivial non-negative consistent heuristic to receive any points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll be graded:

Number of nodes expanded	Grade
more than 2000	0/3
at most 2000	1/3
at most 1600	2/3
at most 1200	3/3

*Remember:* If your heuristic is inconsistent, you will receive no credit, so be careful!

### 1.5 Question 3: Eating All The Dots ( 4 points)

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition which formalizes the food-clearing problem: `FoodSearchProblem` in `searchAgents.py` (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For the present project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next project.) If you have written your general search methods correctly, A\* with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to `testSearch` with no code change on your part (total cost of 7).

`python pacman.py -l testSearch -p AStarFoodSearchAgent` Note: `AStarFoodSearchAgent` is a shortcut for `-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic`.

*Note:* Make sure to complete Question 1 before working on Question 3, because Question 3 builds upon your answer for Question 1.

Fill in `foodHeuristic` in `searchAgents.py` with a consistent heuristic for the `FoodSearchProblem`. Try your agent on the `trickySearch` board:

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

Any non-trivial non-negative consistent heuristic will receive 1/4. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll get additional points:

Number of nodes expanded	Grade
more than 15000	1/4
at most 15000	2/4
at most 12000	3/4
at most 9000	4/4 (full credit; medium)
at most 7000	5/4 (optional extra credit; hard)

## 2 Multi-Agent Pacman

### 2.1 Introduction

In this section, you will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement both minimax and  $\alpha - \beta$  pruning. The code for this section contains the following files, available in `multiagent.zip`.

---

**Files you'll edit:**

<code>multiAgents.py</code>	Where all of your multi-agent search agents will reside.
<code>pacman.py</code>	The main file that runs Pacman games. This file also describes a Pacman GameState type, which you will use extensively in this project
<code>game.py</code>	The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
<code>util.py</code>	Useful data structures for implementing search algorithms.

---

**Files you can ignore:**

<code>graphicsDisplay.py</code>	Graphics for Pacman
<code>graphicsUtils.py</code>	Support for Pacman graphics
<code>textDisplay.py</code>	ASCII graphics for Pacman
<code>ghostAgents.py</code>	Agents to control ghosts
<code>keyboardAgents.py</code>	Keyboard interfaces to control Pacman
<code>layout.py</code>	Code for reading layout files and storing their contents
<code>autograder.py</code>	Project autograder
<code>testParser.py</code>	Parses autograder test and solution files
<code>testClasses.py</code>	General autograding test classes
<code>test_cases/</code>	Directory containing the test cases for each question
<code>multiagentTestClasses.py</code>	Specific autograding test classes

---

**Files to Edit and Submit:** You will fill in portions of `multiAgents.py` during the assignment. You should submit this file with your code and comments. Please do not change the other files in this distribution or submit any of our original files other than this file.

**Evaluation:** Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder.

## 2.2 Multi-Agent Pacman

First, play a game of classic Pacman:

```
python pacman.py
```

Now, run the provided ReflexAgent in multiAgents.py:

```
python pacman.py -p ReflexAgent
```

Note that it plays quite poorly even on simple layouts:

```
python pacman.py -p ReflexAgent -l testClassic
```

Inspect its code (in multiAgents.py) and make sure you understand what it's doing.

## 2.3 Question 4: Minimax ( 5 points)

Now you will write an adversarial search agent in the provided MinimaxAgent class stub in multiAgents.py. Your minimax agent should work with any number of ghosts, so you'll have to write an algorithm that is slightly more general than what you've previously seen in lecture. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.

Your code should also expand the game tree to an arbitrary depth. Score the leaves of your minimax tree with the supplied self.evaluationFunction, which defaults to scoreEvaluationFunction. MinimaxAgent extends MultiAgentSearchAgent, which gives access to self.depth and self.evaluationFunction. Make sure your minimax code makes reference to these two variables where appropriate as these variables are populated in response to command line options.

*Important:* A single search ply is considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving two times.

*Grading:* We will be checking your code to determine whether it explores the correct number of game states. This is the only way reliable way to detect some very subtle bugs in implementations of minimax. As a result, the autograder will be very picky about how many times you call GameState.generateSuccessor. If you call it any more or less than necessary, the autograder will complain. To test and debug your code, run

```
python autograder.py -q q2
```

This will show what your algorithm does on a number of small trees, as well as a pacman game. To run it without graphics, use:

```
python autograder.py -q q2 --no-graphics
```

### *Hints and Observations*

- The correct implementation of minimax will lead to Pacman losing the game in some tests.



This is not a problem: as it is correct behaviour, it will pass the tests.

- The evaluation function for the pacman test in this part is already written (`self.evaluationFunction`). You shouldn't change this function, but recognize that now we're evaluating *\*states\** rather than actions, as we were for the reflex agent. Look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.
- The minimax values of the initial state in the `minimaxClassic` layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively. Note that your minimax agent will often win (665/1000 games for us) despite the dire prediction of depth 4 minimax.

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

- Pacman is always agent 0, and the agents move in order of increasing agent index.
- All states in minimax should be `GameStates`, either passed in to `getAction` or generated via `GameState.generateSuccessor`. In this project, you will not be abstracting to simplified states.
- On larger boards such as `openClassic` and `mediumClassic` (the default), you'll find Pacman to be good at not dying, but quite bad at winning. He'll often thrash around without making progress. He might even thrash around right next to a dot without eating it because he doesn't know where he'd go after eating that dot. Don't worry if you see this behavior, question 5 will clean up all of these issues.
- When Pacman believes that his death is unavoidable, he will try to end the game as soon as possible because of the constant penalty for living. Sometimes, this is the wrong thing to do with random ghosts, but minimax agents always assume the worst:

```
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
```

## 2.4 Question 5: $\alpha - \beta$ Pruning ( 5 points)

Make a new agent that uses  $\alpha - \beta$  pruning to more efficiently explore the minimax tree, in `AlphaBetaAgent`. Again, your algorithm will be slightly more general than the pseudocode from lecture, so part of the challenge is to extend the  $\alpha - \beta$  pruning logic appropriately to multiple minimizer agents.

You should see a speed-up (perhaps depth 3 alpha-beta will run as fast as depth 2 minimax). Ideally, depth 3 on `smallClassic` should run in just a few seconds per move or faster.

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

The `AlphaBetaAgent` minimax values should be identical to the `MinimaxAgent` minimax values, although the actions it selects can vary because of different tie-breaking behavior. Again, the minimax values of the initial state in the `minimaxClassic` layout are 9, 8, 7 and -492 for depths 1, 2, 3 and 4

respectively.

*Grading:* Because we check your code to determine whether it explores the correct number of states, it is important that you perform alpha-beta pruning without reordering children. In other words, successor states should always be processed in the order returned by `GameState.getLegalActions`. Again, do not call `GameState.generateSuccessor` more than necessary.

You must not prune on equality in order to match the set of states explored by our autograder. (Indeed, alternatively, but incompatible with our autograder, would be to also allow for pruning on equality and invoke alpha-beta once on each child of the root node, but this will not match the autograder.)

The pseudo-code below represents the algorithm you should implement for this question.

## Alpha-Beta Implementation

---

$\alpha$ : MAX's best option on path to root  
 $\beta$ : MIN's best option on path to root

```
def max-value(state,  $\alpha$ ,  $\beta$ ):  
    initialize  $v = -\infty$   
    for each successor of state:  
         $v = \max(v, \text{value}(\text{successor}, \alpha, \beta))$   
        if  $v > \beta$  return  $v$   
         $\alpha = \max(\alpha, v)$   
    return  $v$ 
```

```
def min-value(state,  $\alpha$ ,  $\beta$ ):  
    initialize  $v = +\infty$   
    for each successor of state:  
         $v = \min(v, \text{value}(\text{successor}, \alpha, \beta))$   
        if  $v < \alpha$  return  $v$   
         $\beta = \min(\beta, v)$   
    return  $v$ 
```

### 3 Notes

1. The related files written in Python2 can be found in `search.zip` and `multiagent.zip`. You can refer to <https://www.liaoxuefeng.com/wiki/001374738125095c955c1e6d8bb493182103fac9270762a000> if you are a noob of Python.
2. You can ask one partner to join you to do together. Please complete the above five questions, and you only need to send `P01_YourNumber.zip` which contains the modified `search.zip` and `multiagent.zip` to the mailbox([ai\\_2020@foxmail.com](mailto:ai_2020@foxmail.com)) before the deadline(2020/09/27 23:59:59). For this project, you needn't submit the pdf file.
3. Last but not least, you are not alone! If you find yourself stuck on something, contact the TA for help.

## 4 Results

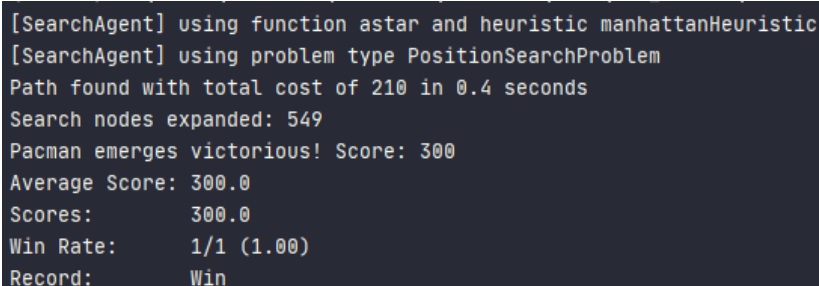
考虑到项目代码均为 python2 代码，我采用的环境是 Pycharm+Python2.7 来完成本次项目，所有代码由本人独立完成，下面将介绍各个子任务的完成情况以及展示相关的运行结果。

### 4.1 Question 1: A\* search

第一个子任务核心在于运用理论课上所学到的 A\* 算法对参数中的 problem 进行求解，需要维护的数据结构包括边界 frontier 和 accessed 两个，frontier 的格式为 (state, actions, fn)，分别表示当前状态、初始状态到当前状态采取的行动序列和当前状态的 f(n) 值，accessed 则只包含 state 当前状态，因为 state 唯一地代表了状态空间中的一种状态，所以 accessed 可用于环检测 CycleChecking，在访问一个节点并进行扩展时，需要判定该节点是否已经在之前的搜索中被扩展过。

下面给出以下命令的执行结果：

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=
manhattanHeuristic
```



```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.4 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:      300.0
Win Rate:    1/1 (1.00)
Record:      Win
```

### 4.2 Question 2: Corners Problem: Heuristic

第二个子任务建立在第一个子任务完成的 A\* 算法基础之上，要求我们完善一个 CornersProblem 的类，解决从当前位置出发找到最优的经过四个顶点的路径，核心在于启发式函数如何设计以及状态空间如何定义。考虑到四个顶点数量比较少，全排列下来也只有 24 种情况，而问题又要求尽可能减小扩展节点的数量，因此我在初始化 CornersProblem 类的时候就利用宽度优先搜索 BFS 对距离直接进行打表，计算地图上所有非 Wall 位置到四个顶点的真实距离，以及每个顶点到其他三个顶点的真实距离，分别设置这两个距离矩阵。

需要先声明的是，我设计的状态空间是坐标点对的基础上增加了一个四元素的 bool 列表，该列表保存了当前状态是否已经访问过四个节点，如初始状态就包含列表 [False, False, False, False]。如此在启发式函数中，就只需要筛选出未到达的所有顶点，然后求出它们的全排列，计算出这些排列中最短的经过所有顶点的路径长度作为返回值即可。

但这样做一个最大的问题是打表是需要一定的时间的，更好的做法应该是采取到四个顶点的曼哈顿距离之和，这个留到第三个子任务再详细讲述，好处在于需要扩展的节点非常非常少，远小于项目要求的 1200 个节点，下面给出以下命令的执行结果：

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

```
Path found with total cost of 106 in 9.8 seconds
Search nodes expanded: 189
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:         434.0
Win Rate:       1/1 (1.00)
Record:         Win
```

```
python autograder.py -q q6
```

```
path length: 106
*** PASS: Heuristic resulted in expansion of 189 nodes

### Question q6: 3/3 ###

Finished at 22:47:39

Provisional grades
=====
Question q4: 3/3
Question q6: 3/3
-----
Total: 6/6
```

### 4.3 Question 3: Eating All The Dots

第三个子任务和第二个子任务本质上没有太大的区别，只是不需要我们自己去完善 `FoodSearchProblem` 类，只需要编写一个启发式函数即可。因为地图上包含很多很多的豆子，再去打表和进行全排列枚举几乎是不可能的，因此我们需要实现一个 `admissible and consistent` 的启发式函数。这里我尝试了不少种组合方法，查阅了网上各种解决方案，最后采用的是较为简洁的一种，计算当前位置到所有豆子的曼哈顿距离最大值，这种方案一定是 `admissible` 的因为 `agent` 最少也要走过这个距离才有可能吃完所有豆子，`consistent` 则可以通过实际运行效果大致检测出来。

最后的运行结果还算不错，扩展了 9551 个节点，下面给出以下命令的执行结果：

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

```
Path found with total cost of 60 in 5.0 seconds
Search nodes expanded: 5423
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:         570.0
Win Rate:       1/1 (1.00)
Record:         Win
```

```
python autograder.py -q q7
```

```
***      expanded nodes: 9551
***      thresholds: [15000, 12000, 9000, 7000]

### Question q7: 3/4 ###

Finished at 22:50:18

Provisional grades
=====
Question q4: 3/3
Question q7: 3/4
-----
Total: 6/7
```

#### 4.4 Question 4: Minimax

第四个子任务和前三个都有所不同，主要是需要面对的情景不同，地图上的 ghost 会进行移动，要求算法能够实时输出结果，这就要求我们实现一个 MiniMax 博弈算法来挑选出下一步合适的移动。然而 MiniMax 算法是我们先前在实验中就有编写过的，该任务相较于 Othello 其实更简单，不需要考虑太多的特殊情况。

然而 multiagent 和 agent 代码结构是有所不同的，最大的难点在于读懂整个 pacman-multiagen 的运行框架，要熟悉 GameState 的一些调用接口和游戏本身 Agent 的编号设计等。在 MiniMax 算法设计方面有一个不同点，那就是最小博弈者是有多多个的，即多个 Ghost 都需要进行移动，每一层 depth 都对应 pacman 和所有 ghost 运动一次，包含一个最大层 MAX 和若干个最小层 MIN，这就要求我们在进行递归调用时注意 depth 和 agentIndex 的设置。

下面给出以下命令的执行结果：

```
python autograder.py -q q2 --no-graphics
```

```
### Question q2: 5/5 ###

Finished at 22:17:01

Provisional grades
=====
Question q2: 5/5
-----
Total: 5/5

Your grades are NOT yet registered. To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

#### 4.5 Question 5: $\alpha - \beta$ Pruning

第五个子任务在第四个子任务上添加 alpha 值和 beta 值就可以实现，在最小层满足  $\beta < \alpha$  和最大层  $\alpha > \beta$  时需要进行剪枝。这里另外提一下我在测试过程中遇到的一个小 bug，我们在理论课上学到的关于  $\alpha$ - $\beta$  剪枝中每一层返回的都是 alpha 值或者 beta 值，而不是当前层遇到的最大值或最小值，这两种方案并不会对最后结果产生影响，但是会对中间部分的剪枝情况造成差异，也正是因为这一点我迟迟没有通过 q3 的测试，经过修改成后一种方案，每一层返回当前层遇到的最大值或最小值，才最终取得正确结果。

下面给出以下命令的执行结果：

```
python autograder.py -q q3 --no-graphics
```

```
### Question q3: 5/5 ###

Finished at 22:22:41

Provisional grades
=====
Question q3: 5/5
-----
Total: 5/5

Your grades are NOT yet registered. To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

#### 4.6 Summary

总结来说，其实只要对理论课上的 A\* 算法和 Alpha-Beta 剪枝两种算法熟练掌握，需要编码的内容并不算多，最大的问题还是在于每个人对于 search 和 multiagent 两份项目代码框架的不熟悉，这

个需要大量的时间去熟悉和探索，我比较喜欢的一种方法是一边看一边在一些函数中输出一些重要的值来看看内部是什么东西，毕竟 python 代码中绝大部分都只需要调用 print 函数就可以轻松调出来看。

除此之外，我认为本次项目中考察比较多的另外一个点是启发式函数的设计，例如第三个子任务中，启发式函数对最后的结果有非常大的影响，扩展的节点数量不同就会导致运行时间的长短差异非常大，这方面可能是今后需要额外的积累经验的地方。像我采用的曼哈顿距离之和就应该算是不错的方案，但应该还有运算更快、扩展结点更少的方案等待大家一起来分享。