[CS303]Practice 3 - Uninformed Search & Practice 4 - Informed Search

叶璨铭，12011404@mail.sustech.edu.cn

Introduction

A **Problem-Solving Agent** is a type of goal-based agent that views the environment as a collection of atomic states. Its objective is to find a sequence of actions that leads from the initial state to the goal state. Such agents can address a wide range of problems, including the 8-queens puzzle, robot navigation, the Traveling Salesman Problem (TSP), and certain mathematical problems. The underlying mechanism is **search**, which can be categorized into:

- **Uninformed Search**
- **Informed Search**
- **Local Search**
- **Adversarial Search** (used in your Project 1!)

In this lab, you will implement a problem-solving agent that searches for a path in a grid map. This is a classical problem you have already encountered in data structures and algorithm analysis.

Practice 3 – Uninformed Search

You are required to implement the following algorithms:

- **Depth First Search (DFS)**
- **Breadth First Search (BFS)**
- **Uniform Cost Search (UCS)**

📌 **Deadline: Oct. 19**

Practice 4 – Informed Search

In the following week, you will extend your implementation to include:

- **Greedy Best-First Search (GBFS)**
- **Astar Search**

📌 **Deadline: Oct. 26**

Evaluation Criteria

Your submission will be assessed on:

- Clear understanding of every line of your code (no blind copy-paste).
- Successful compilation of the program.
- Correctness of the program logic.
- Reasonable efficiency of the solution.

Grading Policy

- **On-time submission:** 60–100 points
- **Late submission:** 0 points

Tasks & Sample Output

Depth First Search

Below is a sample grid map, where the **blue point** is the start position and the **green point** is the goal. The agent's task is to find a path from start to goal.

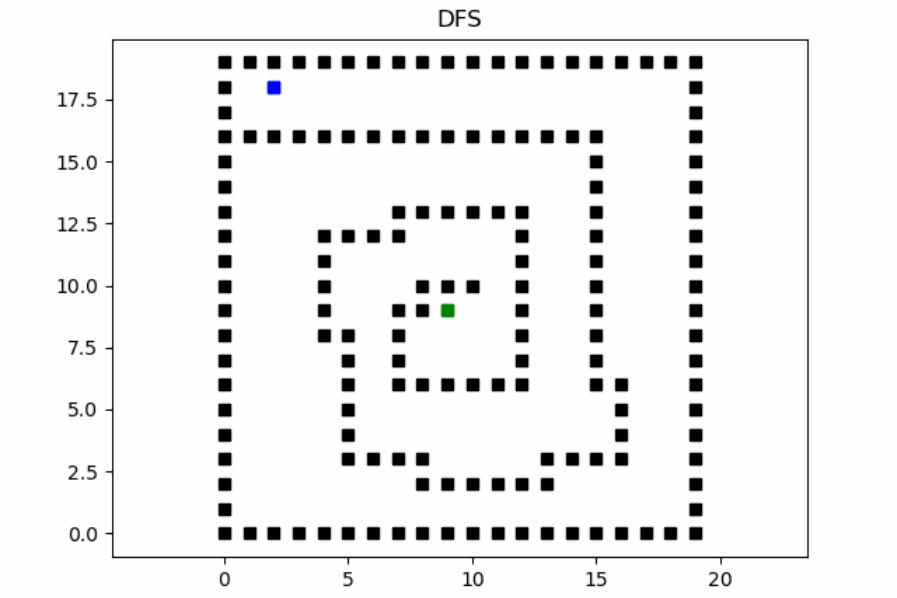
You need to complete the method `DFS` in `agent.py`. Then run `main.py`:

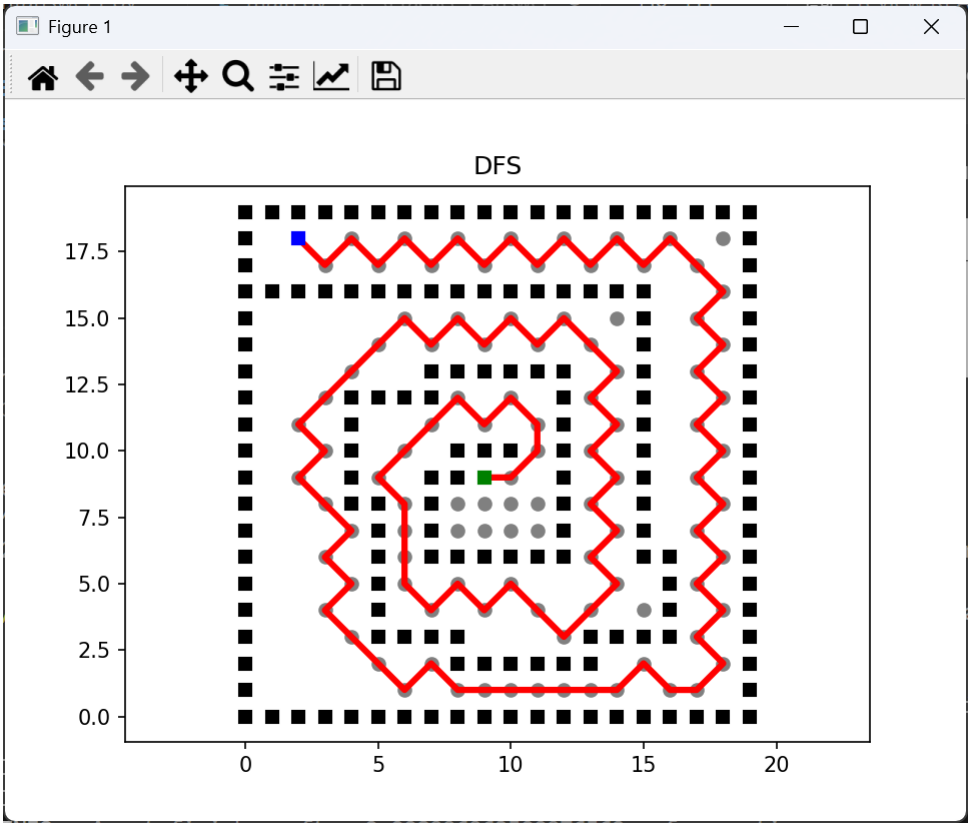
```
python main.py
```

Example interaction:

```
Please input the test case number, or input q to quit. 1
Please choose an algorithm to display, or input c to checkout another map, or
input d to enable or disable gif saving.
Supported algorithms are ['DFS', 'BFS', 'UCS', 'GBFS', 'Astar']DFS
```

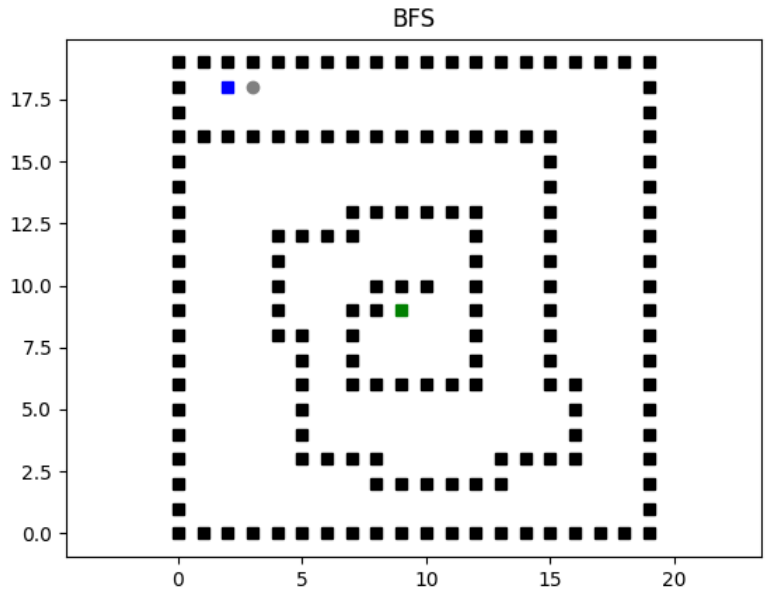
Choosing **test case 1** with **DFS** should give the following results:

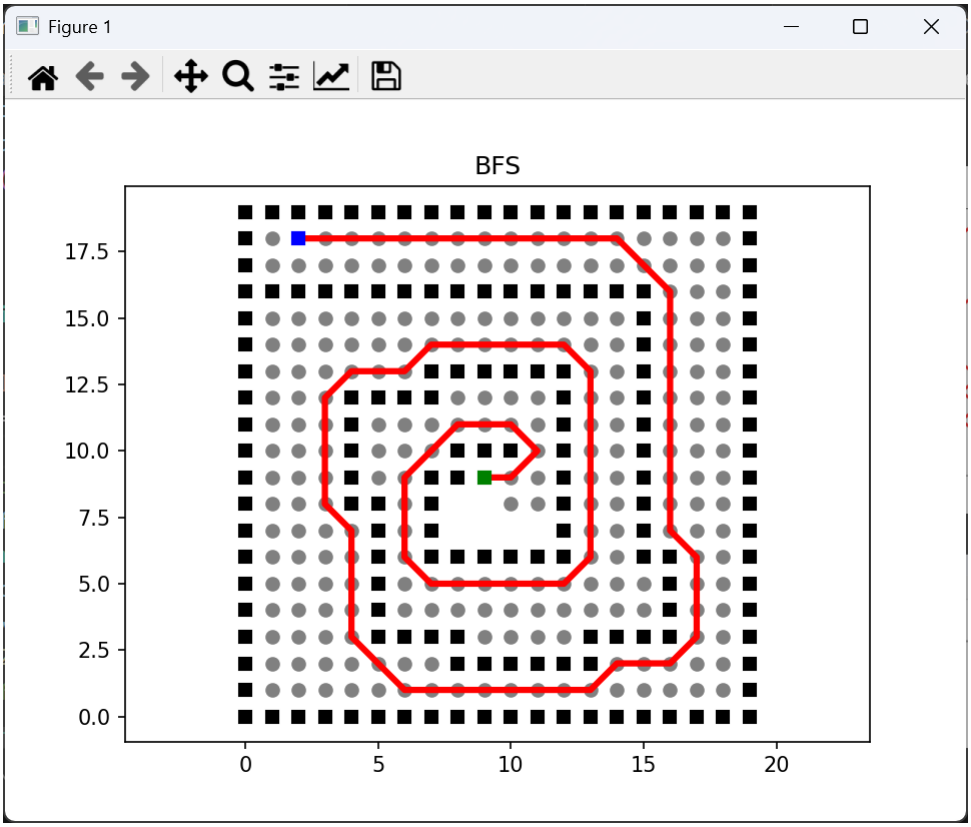




Breadth First Search

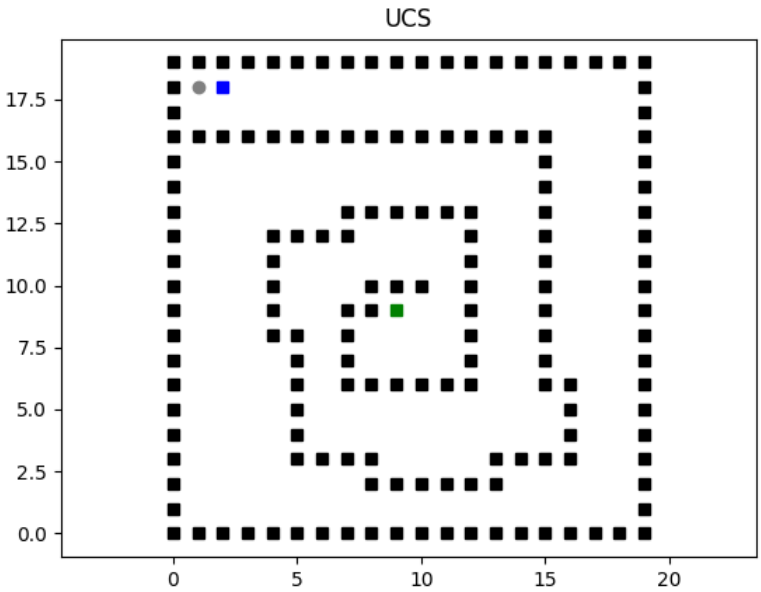
Similarly, complete the `BFS` method in `agent.py`. The output will look like this:

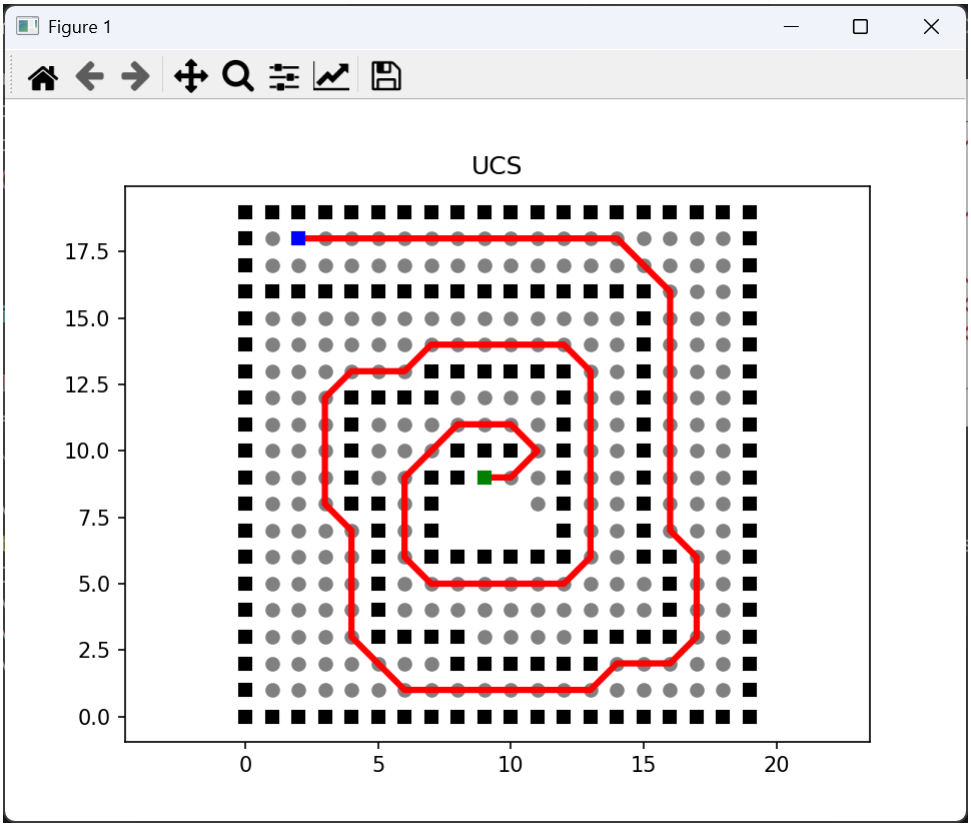




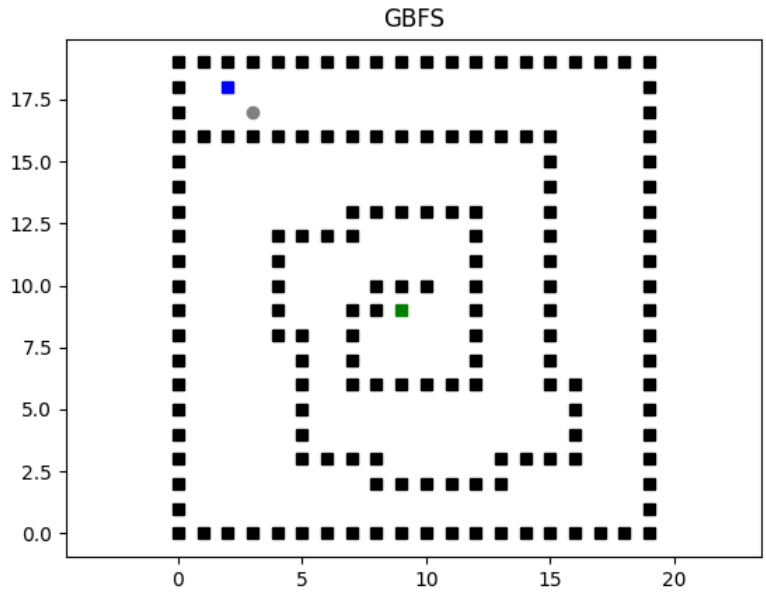
Uniform Cost Search

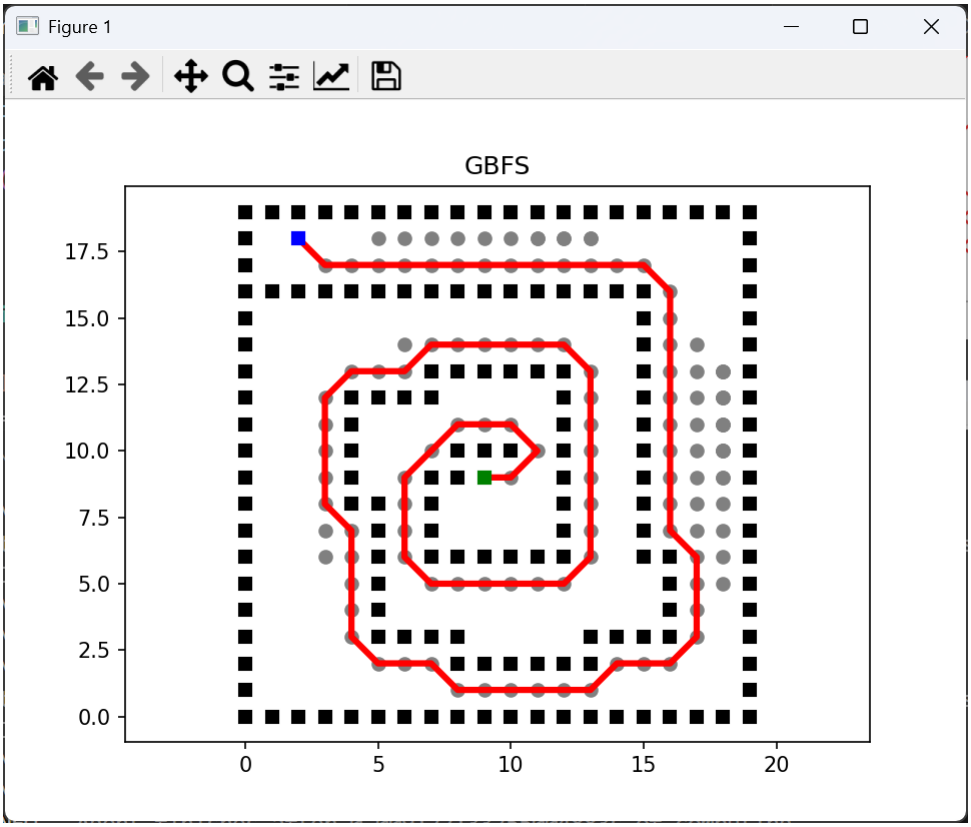
Next, complete the `UCS` method in `agent.py`. Expected results:



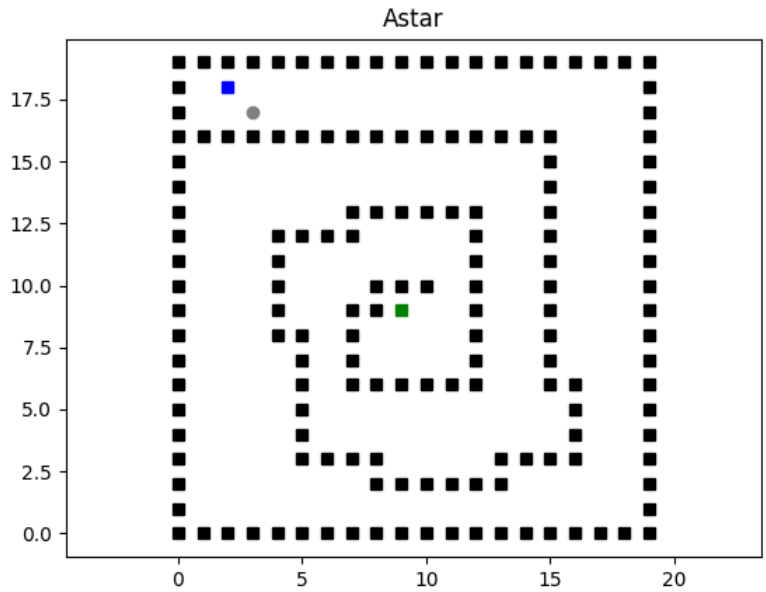


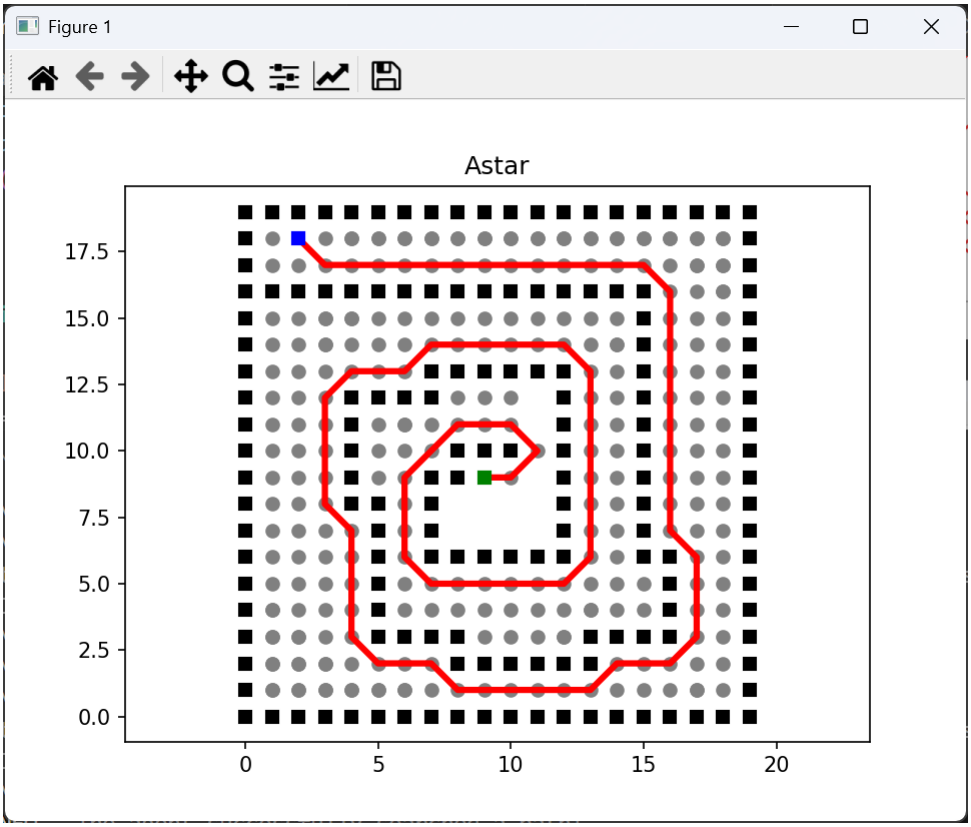
Greedy Best-First Search





Astar Search





Coding Instruction

The project directory has the following structure:

```
├──gif
├──images
├──test_cases
├──utils
├──agent.py
├──main.py
```

```
├──plotting.py
├──requirements.txt
```

The Project is composed of 4 folders and 4 files.

- `requirements.txt` → Install all required dependencies before running `main.py` by executing:

  ```
  pip install -r requirements.txt
  ```

- `agent.py` → main file to implement search algorithms.
- `main.py` → runs and tests your implementation.
- `plotting.py` & `utils` & `images` & `gif` → helper functions for visualization.
- `test_cases` → configuration files (`.toml`) that describe maps and obstacles (similar to JSON but more human-friendly).

Key Implementation: `agent.py`

Inside class `ProblemSolvingAgent`, you will find the following method:

```
def solve_by_searching(self, obstacles, start_pos, goal_pos, algorithm='DFS'):
```

This is the **API** invoked by `main.py`. It dispatches tasks to different search methods (DFS, BFS, UCS, etc.), so you don't need to modify it.

You must complete the following methods:

```
def DFS(self, obstacles, start_pos, goal_pos):
    path, visited = [], []
    return path, visited
def BFS(self, obstacles, start_pos, goal_pos):
    path, visited = [], []
    return path, visited
def UCS(self, obstacles, start_pos, goal_pos):
    path, visited = [], []
    return path, visited
def GBFS(self, obstacles, start_pos, goal_pos):
    path, visited = [], []
    return path, visited
def Astar(self, obstacles, start_pos, goal_pos):
    path, visited = [], []
    return path, visited
```

The Python docstring of `solve_by_searching` explains the data types and meanings of its input and output variables. Be sure to go through it carefully before you start using the function.

```
"""Let the agent solve problem by searching path on the graph.
        Args:
            obstacles (list of bi-tuples):
                Obstacles represents the graph information of the grid map,
                by a list of points called obstacles.
                At any coordinate, you are allowed to move to
                any node nearby that is not in the obstacles.
                When coding, you can use self.neighbours(obstacles, node)
            start_pos (bi-tuples): the position of initial state.
            goal_pos (bi-tuples): the position of goal state.
            algorithm (str, optional): The strategy applied by the agent.
                Defaults to 'DFS'.
        Returns: tuple (path, visited)
            path (list of bi-tuples): the path chosen by the algorithm
                to navigate from initial position to the goal position
            visited(list of bi-tuples): the position checked by the agent
                during the searching process.
        """
```

We also provide several helper APIs to assist your programming. The most important one is `neighbours_of`, which is used to retrieve all valid neighboring nodes of a given position in the grid map, along with their corresponding movement costs. This function is analogous to accessing the adjacency list of a graph.

```
def neighbours_of(self, obstacles, node):
        """_summary_

        Args:
            obstacles (_type_): _description_
            node (_type_): _description_
        Returns: iterable generator of tuple(neighbour, moving_cost)
            neighbour(bi-tuple): a position near to the node.
            moving_cost(float): the cost the agent has to pay to move from node to
neighbour.
        """
```

```
# Example: 2x2 grid map, Obstacles: [(0,1)]
#(0,0)  S   |  (0,1)  •
#------------------------
#(1,0)  X   |  (1,1)  •
self.neighbours_of(obstacles, (0, 0)) == [((1,1), sqrt(2)), ((1,0), 1)]
```

Frequently asked questions

UCS looks like Dijkstra learnt in DSAA course, what 's the difference?

Basically, they are the same algorithm viewed from different perspectives. In my opinion,

- On one hand, Dijkstra studies the single source shortest path problem. He tries to answer why inspecting the node in the order of UCS makes the path shortest.
- On the other hand, UCS is emphasizing the searching priority should be the cost from the initial state, which means each decisions made by the agent is uniform, or in other words, fair. This perspective emphasizes that the agent is searching with no extra information about the goal, but only with the knowledge of what it has done and what cost it has taken.

Felner points out in his paper that the original description of Dijkstra algorithm is not efficient, and UCS usually accepts the efficient implementation with priority queue. See reference[^3] (the pdf file in lab folder) for more details.

What is "Uninformed Search"

Uninformed Search is said compared to Informed Search. Recall that when we do searching, we only knows how to transfer from one atomic state to another, states are not related to each other. But in the grid map case, we know the physical meaning of two states, that their attributes are the x-y locations in the coordinate system. Then we don't treat states as atomic, instead we treat them as two coordinates. We can then construct "Informed Search" algorithms, such as "Greedy Best First Search" or "A star Search" to improve the search speed, which is going to be taught next week.

Is BFS the same as UCS in the grid map?

No. BFS searches in an first in first out order, while UCS searches in the order of distance from the starting point. In this lab, we consider moving from (0, 0) to (1,1) to cost sqrt(2), not 1. So you should see a circle by running UCS, while seeing a square by running BFS.

References

[1] zhm-real, *PathPlanning: Common path planning algorithms with animations*.
Available: https://github.com/zhm-real/PathPlanning, accessed Sep. 21, 2022.

[2] Intelligent-Robot-Course, *Homework repository*.
Available: https://github.com/Intelligent-Robot-Course/Homework, accessed Sep. 21, 2022.

[3] A. Felner, “Dijkstra’s Algorithm versus Uniform Cost Search or a Case Against Dijkstra’s Algorithm,” Position Paper, 2005.

[4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*,
3rd ed., Beijing: Tsinghua University Press, 2013. ISBN: 9787302331094.