Artificial-Intelligence-Lab3

叶璨铭，12011404@mail.sustech.edu.cn

1 Introduction

Problem Solving Agent is a kind of goal-based agent who treat the environment as atomic states. The goal of the Problem Solving Agent is to find a sequence of actions that will lead to the goal state from the initial state. Problem Solving Agent can solve many problems, including 8-queens problem, robot navigation problem, TSP problem and some math problems. Problem Solving Agent solves these problems by searching, which can be grouped into Uninformed Search, Informed Search, Local Search, Adversarial Search(which is used to implement your Project 1 !).

In this lab, you are going to implement a problem solving agent that searches path in a grid map. This is a classical problem that you have learnt in the perspective of data structure and algorithm analysis.

2 Lab Practice

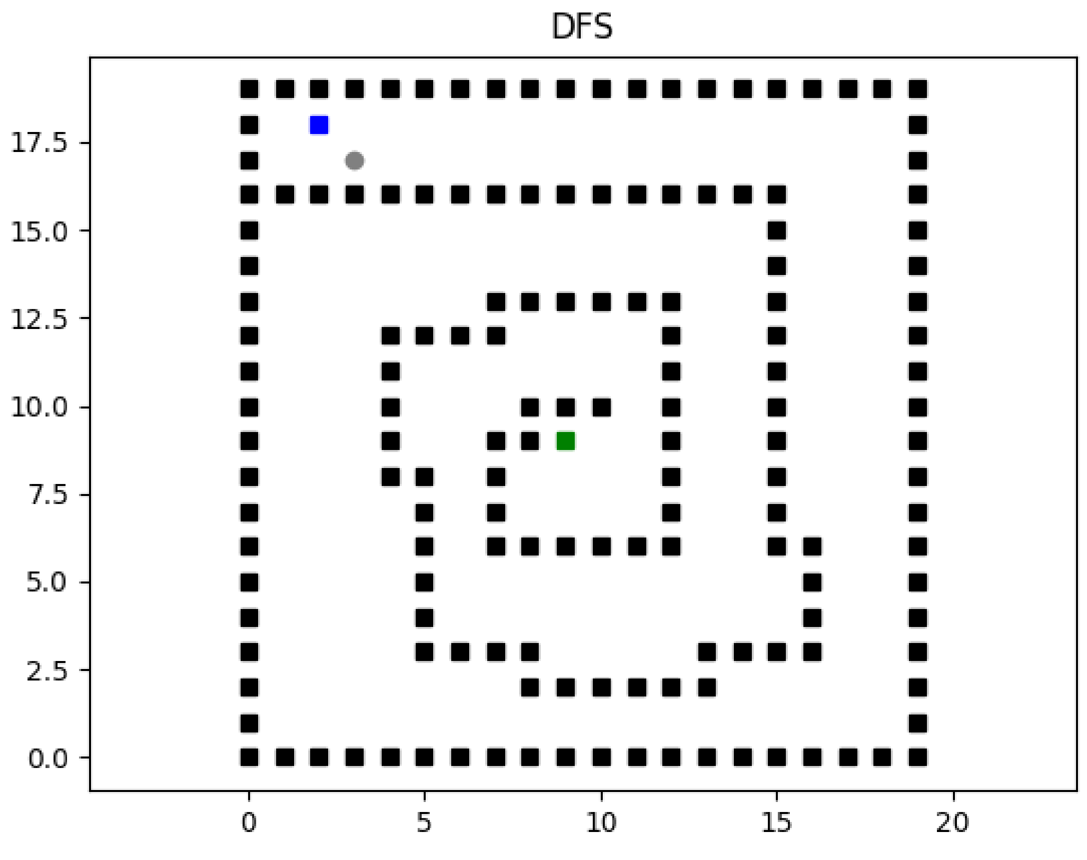
In Lab3, you are going to implement 3 algorithms, DFS, BFS and UCS.

In Lab4 next week, we are going to go through GBFS(Greedy best first search) and A star.

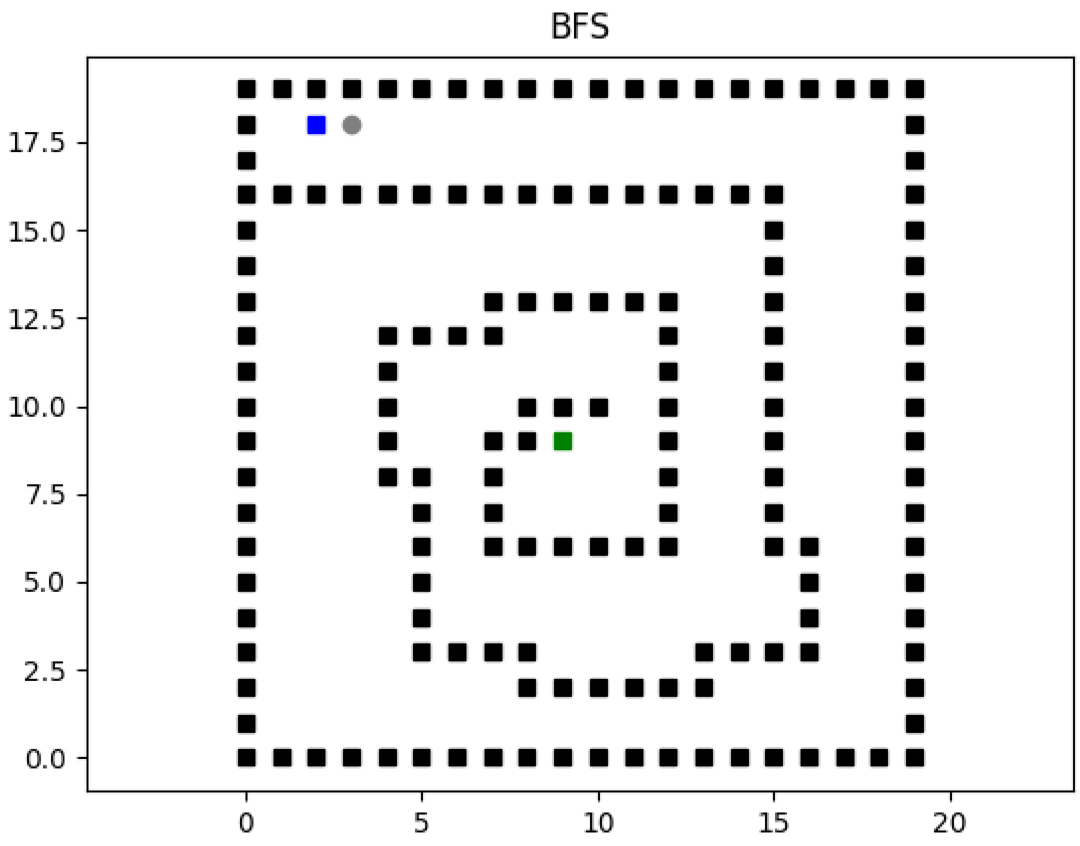
2.1 Depth First Search

Here is a map, where the blue point is the starting point, and the green point is the goal point. The problem for the agent to solve is to find a path that reaches the goal state.

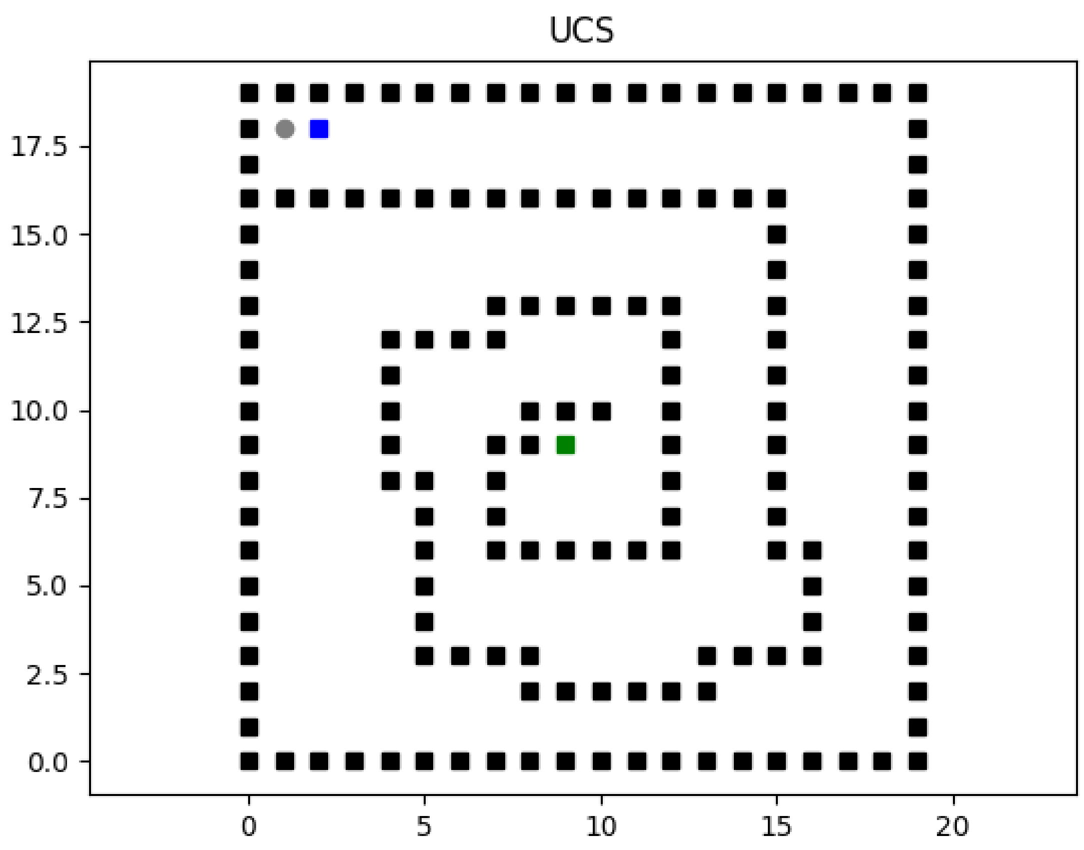
You need to complete the method `DFS` in `agent.py`. The you can see the effect like this:



2.2 Breadth First Search



2.3 Uniform Cost Search



3 Coding Instruction

```
├──gif
├──test_cases
├──utils
├---agent.py
├---main.py
├---plotting.py
```

The Project is composed of 3 folders and 3 files.

- The main part you should finish is `agent.py` , in which the path searching algorithms will be implemented.
- If you finishes the algorithms, you can run `main.py` to test your program.
- `plotting.py` and `utils` contains some tools to help main.py and agent.py to visualize the running process of the algorithm you implemented.
- `test_cases` includes world configuration files suffixed after `toml`, a popular markup language that is equivalent to `json` but is more human friendly. The `main.py` will read different test cases configuration and load different maps described by `.png` file to test your algorithms.

Now let's look deeper into `agent.py`

3.1 agent.py

In class `ProblemSolvingAgent` you can see this method.

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

The method is the API of the agent, which will be invoked by the `main.py`.

`solve_by_searching` will dispatch the task to different methods below, so you don't need to change it.

What you need to do is to 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 UFS(self, obstacles, start_pos, goal_pos):
    path, visited = [], []
    return path, visited
```

The data types and meanings of the input output variables are described in the Python doc of `solve_by_searching`. Read them carefully!

```
   """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.
          """
```

There is also some APIs to help your programming, among which they most important one is `neighbours_of`

```
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.
        """
```

This is the same as `adj()` of a graph represented by `adjecency list` you learnt in `DSAA course`. For example,

```
# 2x2 grid map graph is (0, 0)<-sqrt(2)->(1,1)
(0,0)<-1->(1, 0) (1, 1)<-1->(1,0)
# obstacles are [(0, 1)]
self.neighbours_of(obstacles, (0, 0)) == [((1,1), sqrt(2)), ((1,0), 1)]
```

4 Frequently asked questions

4.1 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 [1] (the pdf file in lab folder) for more details.

4.2 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.

4.3 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.

Reference

1. A. Felner, “Position Paper: Dijkstra’s Algorithm versus Uniform Cost Search or a Case Against Dijkstra’s Algorithm,” p. 5. ↩