Lab 1: Getting Started & Search

519021910702 游克垚

1 Exercise 1: Stack & Queue

1.1 Stack

Stack is a data structure that manage the data based on the principle of last in first out. My implementation is as follows:

```python
class Stack:
    def __init__(self):
        self._list = []

    def push(self, item):
        self._list.append(item)

    def pop(self):
        return self._list.pop()

    def is_empty(self):
        return len(self._list) == 0
```

1.2 Queue

Queue is a data structure that manage the data based on the principle of first in first out. My implementation is as follows:

```python
class Queue:
    def __init__(self):
        self._list = []

    def push(self, item):
        self._list.append(item)

    def pop(self):
        return self._list.pop(0)

    def is_empty(self):
        return len(self._list) == 0
```

2 Exercise 2: Depth first search

In depth first search, we use a stack to maintain the fringe, which means we always explore the node that has just been added to the fringe so that we can expand a deepest node first. My implementation is as follows:

```python
def depth_first_search(problem):
    from util import Stack

    explored = []
    s = Stack()
    s.push((problem.get_start(), []))

    while True:
        if s.is_empty():
            return []

        now = s.pop()
        explored.append(now[0])
        if problem.is_goal(now[0]):
            return now[1]
        successors = problem.get_successors(now[0])
        for succ in successors:
            if not succ[0] in explored:
                s.push((succ[0], now[1] + [succ[1]]))
```

In order to remember the route, I push the state and the corresponding route as a tuple into the stack so that the answer can be returned immediately after the goal is found. The list **explored** is used to store the explored node. The result is as follows:

```
Result:
▯ Successfully reach the goal (cost=10).
```

3 Exercise 3: Breadth first search

In breadth first search, we use a queue to maintain the fringe, which means we always explore the node that was first added to the fringe so that we can expand a shallowest node first.

```python
def breadth_first_search(problem):
    from util import Queue

    explored = []
    s = Queue()
    s.push((problem.get_start(), []))

    while True:
```

```
        if s.is_empty():
            return []

        now = s.pop()
        explored.append(now[0])
        if problem.is_goal(now[0]):
            return now[1]
        successors = problem.get_successors(now[0])
        for succ in successors:
            if not succ[0] in explored:
                s.push((succ[0], now[1] + [succ[1]]))
```

The algorithm is similar to the depth first search except that it use a queue to maintain the fringe. It also use a list **explored** to store the explored node. The result is as follows:

```
Result:
□ Successfully reach the goal (cost=8).
```

4 Exercise 4: Improvement

Considering I have stored the visited node in each loop, so I can't find a maze world where the algorithm will fail except that there is no way between G and S. So my designed maze is as follows:

```
%%%%%%%%%%%
%  G % S  %
%%%%%%%%%%%
```