Lab 2: Informed Search Methods

519021910702 游克垚

1 Exercise 1

1.1 Number of 'Lakes'

Since a lake a is a connected area of characters %, we can use DFS to traverse the whole lake. So when we find a new lake which has not been explored, increase the number of lakes and do a DFS from here to add the whole lake into the explored set. Do this until all lakes are explored. My implementation is as follows:

```
def Exercise2_1_1(problem):
    from util import Stack

    lakes = 0
    explored = []
    blank = Stack()
    lakes = Stack()
    blank.push(problem.get_start())

    while True:
        if blank.is_empty():
            return lakes

        now_blank = blank.pop()
        explored.append(now_blank)
        blank_successors = problem.get_successors(now_blank)
        lake_successors = problem.get_successors(now_blank, 1)

        for succ in blank_successors:
            if not succ[0] in explored:
                blank.push(succ[0])
        for succ in lake_successors:
            if not succ[0] in explored:
                lakes += 1
                lakes.push(succ[0])
                while not lakes.is_empty():
                    now_lake = lakes.pop()
                    explored.append(now_lake)
```

```
29                      successors = problem.get_successors(now_lake, 1)
30                      for s in successors:
31                          if not s[0] in explored:
32                              lakes.push(s[0])
```

1.2 DFGS with Iterative Deepening

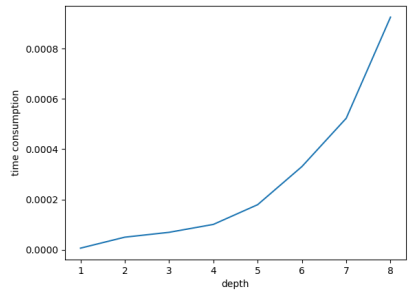
My implementation of DFGS with Iterative Deepening is as follows:

```
1  def Exercise2_1_2(problem):
2      from util import Stack
3
4      depth = 0
5      flag = True
6      while flag:
7          flag = False
8          depth = increase_depth(depth)
9          explored = []
10         s = Stack()
11         s.push((problem.get_start(), [], 0))
12
13         while True:
14             if s.is_empty():
15                 break
16
17             now = s.pop()
18             explored.append(now[0])
19             if problem.is_goal(now[0]):
20                 return now[1]
21
22             if now[2] + 1 < depth:
23                 successors = problem.get_successors(now[0])
24                 for succ in successors:
25                     if not succ[0] in explored:
26                         s.push((succ[0], now[1] + [succ[1]], now[2] + 1))
27             else:
28                 flag = True
```

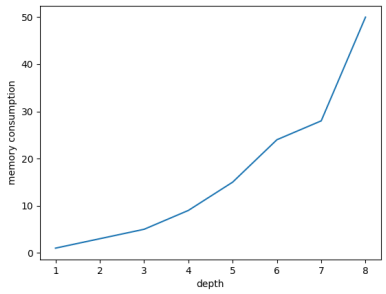
The function increase_depth() is the increasing function on depths. And my two increasing functions is as follows:

```
1  def increase_depth1(depth):
2      return depth + 1
3
4  def increase_depth2(depth):
5      return depth + 2
```

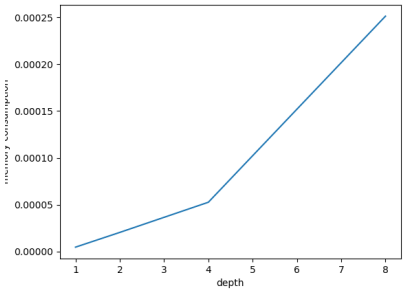
The performances of these two functions are as follows:



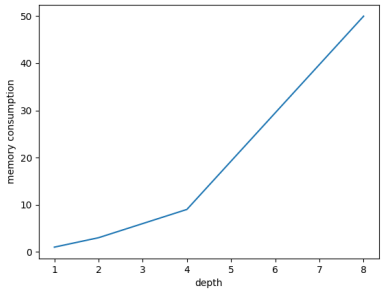
(a) time consumption of increase_depth1



(b) memory consumption of increase_depth1



(c) time consumption of increase_depth2



(d) memory consumption of increase_depth2

Figure 1-1 performances of two increasing functions on depth

The function increase_depth2 uses less time because it skips some depths. In each depth, the memory consumption of these two functions is the same. The advantage of the function increase_depth1 is that it can find the smallest depth to get the result but it will cost more time. The advantage of the function increase_depth2 is that it can quickly find the result. So when we want to find the smallest depth, we can use the increase_depth1. When we want to get the result quickly, we can use the increase_depth2.

2 Exercise 2: Least-Cost Path

2.1 uniform-cost graph search (UCGS)

UCGS is to use a priority-queue whose priority is node's cost to maintain the fringe so that we can expand a cheapest node first. My implementation is as follows:

```
1  def Exercise2_2_1(problem):
2      from queue import PriorityQueue
3
4      count = 0
5      explored = []
6      s = PriorityQueue()
7      s.put((0, count, problem.get_start(), []))
```

```
8
9       while True:
10          if s.empty():
11              return []
12
13          now = s.get()
14          explored.append(now[2])
15          if problem.is_goal(now[2]):
16              return (now[0], now[3])
17          successors = problem.get_successors(now[2])
18          for succ in successors:
19              if not succ[0] in explored:
20                  count += 1
21                  s.put((now[0] + succ[2], count, succ[0], now[3] + [succ[1]]))
```

Cost, count, state and path are pushed into the priority-queue as a tuple. The member count is to solve the problem that one state is put into the queue twice and the path can't be compared. The result on the maze Maze_lab2_2_1.lay is as follows:

```
1 Result:
2 (17, [<Direction.NORTH: 'North'>, <Direction.NORTH: 'North'>, <Direction.NORTH: 'North'
      >, <Direction.NORTH: 'North'>, <Direction.NORTH: 'North'>, <Direction.NORTH: 'North
      '>, <Direction.EAST: 'East'>, <Direction.EAST: 'East'>, <Direction.EAST: 'East'>, <
      Direction.EAST: 'East'>, <Direction.EAST: 'East'>, <Direction.EAST: 'East'>, <
      Direction.EAST: 'East'>, <Direction.EAST: 'East'>, <Direction.EAST: 'East'>, <
      Direction.EAST: 'East'>, <Direction.EAST: 'East'>])
```

2.2 greedy graph search (GGS)

GGS is to use a priority-queue whose priority is node's heuristic value to maintain the fringe so that we can expand the node that seems closest to the goal. My implementation is as follows:

```
1  def Heuristic1(state1, state2):
2      return math.sqrt(math.pow(state1[0]-state2[0], 2)+math.pow(state1[1]-state2[1], 2))
3
4  def Exercise2_2_2(problem):
5      from queue import PriorityQueue
6
7      count = 0
8      explored = []
9      s = PriorityQueue()
10     s.put((Heuristic1(problem.get_start(), problem.get_goal()), count, problem.
           get_start(), []))
11     while True:
12         if s.empty():
13             return []
14         now = s.get()
```

```
15          explored.append(now[2])
16          if problem.is_goal(now[2]):
17              return now[3]
18          successors = problem.get_successors(now[2])
19          for succ in successors:
20              if not succ[0] in explored:
21                  count += 1
22                  s.put((Heuristic1(succ[0], problem.get_goal()), count, succ[0], now[3]
                        + [succ[1]]))
```

Heuristic value, count, state and path are pushed into the priority-queue as a tuple. The result on the maze Maze_lab2_2_1.lay is as follows:

```
1  Result:
2  [<Direction.EAST: 'East'>, <Direction.EAST: 'East'>, <Direction.EAST: 'East'>, <
       Direction.EAST: 'East'>, <Direction.EAST: 'East'>, <Direction.NORTH: 'North'>, <
       Direction.EAST: 'East'>, <Direction.NORTH: 'North'>, <Direction.EAST: 'East'>, <
       Direction.NORTH: 'North'>, <Direction.EAST: 'East'>, <Direction.EAST: 'East'>, <
       Direction.EAST: 'East'>, <Direction.NORTH: 'North'>, <Direction.NORTH: 'North'>, <
       Direction.NORTH: 'North'>, <Direction.EAST: 'East'>]
```

The result is equal to the UCGS on this maze. But GGS is quicker than UCGS. For those environments that there is no obstacle between the start state and the goal state, GGS has better performance. For examples, the following maze is better for GGS.

```
1  %%%%%%%%%%%%%%
2  %            G%
3  %             %
4  %             %
5  %     S       %
6  %             %
7  %             %
8  %             %
9  %%%%%%%%%%%%%%
```

However, for those environments that we can't go straight from the start state to the goal state, UCGS has better performance because its path's cost is least. For example, the following maze is better for UCGS.

```
1  %%%%%%%%%%%%%%
2  %            G%
3  % %%%%%%%%%  %
4  %          %  %
5  %          %  %
6  %          %  %
7  %          %  %
8  %S            %
9  %%%%%%%%%%%%%%
```

Though UCGS can find the least-cost path, it spends much time to find the path, which is its biggest drawback.

3 Exercise 3: Least-Cost Path with Heuristics

A^* graph search is to use a priority-queue whose priority is node's cost and node's heuristic value to maintain the fringe so that we can expand the node that seems close to the goal and has cheap cost. My implementation is as follows:

```
1  def Exercise2_3_1(problem):
2      from queue import PriorityQueue
3
4      count = 0
5      explored = []
6      s = PriorityQueue()
7      s.put((Heuristic1(problem.get_start(), problem.get_goal()), count, problem.
           get_start(), [], 0))
8
9      while True:
10         if s.empty():
11             return []
12
13         now = s.get()
14         explored.append(now[2])
15         if problem.is_goal(now[2]):
16             return (now[4], now[3])
17
18         successors = problem.get_successors(now[2])
19         for succ in successors:
20             if not succ[0] in explored:
21                 count += 1
22                 g = now[4] + succ[2]
23                 h = Heuristic1(succ[0], problem.get_goal())
24                 s.put((g + h, count, succ[0], now[3] + [succ[1]], g))
```

Order value, count, state, path and cost are pushed into the priority-queue as a tuple. The results of UCGS, GGS and A^*GS on the mazes Maze_lab2_2_1.lay and Maze_lab2_3_1.lay are shown in table 3-1 and table 3-2.

	cost	optimal	expanded nodes
UCGS	17	true	15847
GGS	17	true	19
A^*GS	17	true	10244

Table 3-1 The experimental results of UCGS, GGS and A^*GS on the maze Maze_lab2_2_1.lay

	cost	optimal	expanded nodes
UCGS	17	true	2043
GGS	21	false	43
A^*GS	17	true	2024

Table 3-2 The experimental results of UCGS, GGS and A^*GS on the maze Maze_lab2_3_1.lay

For the heuristic function $dis(P,G) = |x_P - x_G| + |y_P - y_G| - \mathbb{I}\{|x_P - x_G| \neq |y_P - y_G|\}$, A^*GS still works under it. The proof is as follows.

$$\begin{aligned} \because \quad & h^*(P) \geq |x_P - x_G| + |y_P - y_G| \\ & dis(P,G) \leq |x_P - x_G| + |y_P - y_G| \\ \therefore \quad & 0 \leq dis(P,G) \leq h^*(P) \end{aligned}$$

So the heuristic value is admissible and A^*GS is optimal. It is also better than the Euclidean distance. Because the expanded nodes of this function on the Maze_lab2_3_1.lay is 1524 which is better than before.