lab1 report

author : 李卓壕

ID : 519021911248

exercise 1

To implement "Stack" and "Queue" is quite simple by using Python3 btf(built-in-functions). Further speaking, we can use "_List" module.

Further speaking, **list** is a built in data structure in Python3, we can implement **push(item)** by using btf **list.append(item)**, which is efficient adding the item at the end of the list.

Also, considering the implement of **pop()**, we can easily use btf **list.pop()**, return the last item in the list.

And, I reload **len()** to finish **is_empty()**, if length=0, then I return true, else return false.

the brief code is as below:

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

    def push(self, item):
        self._list.append(item)
    # using list btf: append(item) to add the item at the last position of the

    def pop(self):
        return self._list.pop()
    # using list btf: pop() to delete the item at the last position of the list

    def is_empty(self):
        return len(self._list)==0
    # using list btf: len() to identify whether the list is empty or not
```

The queue is quite similar to the stack, the only difference between them is implementing **pop()** by using **list_pop(0)** as below:

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

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

    def pop(self):
        return self._list.pop(0)
    # the only difference is, pop(0), which can delete the first item in the li


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

I have tested the data structure by running my own test.py, it is correct.

exercise 2 & 3

1. DFS

From the pseudo code on the course slides, we can derive the Python code. It is implemented since we can fully understand it!

△ DFS's logic is simple:

1. initial a stack, and push the start state into it.
2. using "marked" list to restore the item we have been paced.
3. if the stack isn't empty, pop the last state of the stack.(This implement LIFO) else, return False
4. if the state popped **is** the goal, return the path it paced
5. if not, get the successor of the state, actually, the successor is a tuples, and each of the element is a 3-element tuples.

the 5th step is the core I think, the segment code is below :

```
tuples = problem.get_successors(state)

for i in range(len(tuples)):
    a, *_ = tuples[i]
    state = a

#  tuples restore the popped state's successor, which is a tuples, traverse the tup
```

6. refresh the state by next_state, if the state is not in marked, we can push it into the stack, and push it into the marked list.
7. loop above!

```
'''pre-requisite:
including choosing the right data structure:'''
s = Stack()
initial_state = problem.get_start()
s.push(initial_state)

'''initialize stack with start state by pushing it into the stack'''
marked = [initial_state]

'''mark nodes that have been tested before'''
parent = []

while s.is_empty() == False:
    state = s.pop()
    parent.append(state)
    if problem.is_goal(state):
        return parent
    else:
        tuples = problem.get_successors(state)
        # tuples is ((X,x,x),(X,x,x),(x,x,x),...)
        for i in range(len(tuples)):
            a, *_ = tuples[i]
            state = a
            if state not in marked:
                marked.append(state)
                s.push(state)
else:
    return None
```

the result is correct

2. BFS

Like DFS, the BFS is quite similar to it. From the pseudo code on the slides, we can derive the Python code.

△ BFS's logic:

1. initial a queue, and push the start state into it.
2. using "marked" list to restore the item we have been paced.
3. if the queue isn't empty, pop the first state of the queue. (This implement FIFO) else, return False
4. if the state popped **is** the goal, return the path it paced
5. if not, get the successor of the state, actually, the successor is a tuples, and each of the element is a 3-element tuples.
6. refresh the state by next_state, if the state is not in marked, we can push it into the stack, and push it into the marked list.
7. loop above!

the brief code shows as blow:

```
q = Queue()
initial_state = problem.get_start()
q.push(initial_state)
marked = [initial_state]

while q.is_empty() == False:
    state = q.pop()
    print(state)
    if problem.is_goal(state):
        return None
    else:
        tuples = problem.get_successors(state)
        for i in range(len(tuples)):
            a, *_ = tuples[i]
            state = a
            if state not in marked:
                marked.append(state)
                q.push(state)

else:
    return None
```

exercise 4

Design a maze just like that in 'testMaze.lay'

DFS prefer the left branch in the searching tree, it visits nodes of graph **depth wise**. It visits nodes until reach a leaf nor a node which doesn't have non-visited nodes.Exploration of a node is suspended as soon as another unexpected is found. Using stack data structure to store Unexplored nodes. DFS is faster and require less memory.

In comparison, BFS visits node level by level in graph, A node is fully explored before any other can begin. Uses Queue data structure to store Unexplored nodes. BFS is slower and require more memory.

When there is a loop, DFS might fail. Specially speaking, if the searching tree has a left branch which forms a loop, DFS will continue this branch and will never end. This conclude a infinite loop, which leads the DFS to fail. But in comparison, if BFS is searching the same tree mentioned before, the result can be correct because the algorithm is searching the goal layers to layers.

There is an example of it, show as belows:

```
testMaze.lay


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

DFS will fail if tested on the graph above. But BFS will success.