**程序报告**

学号：2011937 姓名：姜志凯

1. **问题重述**

（简单描述对问题的理解，从问题中抓住主干，必填）

====================================================================

**使用python语言，分别实现基于基础搜索算法（对于迷宫游戏，常见的三种的搜索算法有广度优先搜索、深度优先搜索和最佳优先搜索（A\*)）和Deep QLearning算法的机器人，完成机器人走迷宫。在任意位置可执行的动作包括上“u”、下“d”、左“l”、右“r”，执行不同动作后根据不同的情况会获得不同的奖励，具体情况分为：撞墙、到出口、其余情况。**

**模板中有各种类可以使用，包括迷宫类maze、Robot类QRobot、用于机器人的训练和可视化的类Runner等，可以使用这些类的基础功能，自己实现搜索算法即可。**

**QLearning：Q-Learning 是一个值迭代（Value Iteration）算法。**

**与策略迭代（Policy Iteration）算法不同，值迭代算法会计算每个”状态“或是”状态-动作“的值（Value）或是效用（Utility），然后在执行动作的时候，会设法最大化这个值。**

**因此，对每个状态值的准确估计，是值迭代算法的核心。**

**通常会考虑最大化动作的长期奖励，即不仅考虑当前动作带来的奖励，还会考虑动作长远的奖励。**

1. **设计思想**

（所采用的方法，有无对方法加以改进，该方法有哪些优化方向（参数调整，框架调整，或者指出方法的局限性和常见问题），伪代码，理论结果验证等… **思考题，非必填**）

====================================================================

1. **前期准备：**

**·创建迷宫maze=Maze(maze\_size=10)：主要成员方法：**

**sense\_robot() ：获取机器人在迷宫中目前的位置。**

**return：机器人在迷宫中目前的位置。**

**move\_robot(direction) ：根据输入方向移动默认机器人，若方向不合法则返回错误信息。**

**direction：移动方向, 如:"u", 合法值为： ['u', 'r', 'd', 'l']**

**return：执行动作的奖励值**

**can\_move\_actions(position)：获取当前机器人可以移动的方向**

**position：迷宫中任一处的坐标点**

**return：该点可执行的动作，如：['u','r','d']**

**is\_hit\_wall(self, location, direction)：判断该移动方向是否撞墙**

**location, direction：当前位置和要移动的方向，如(0,0) , "u"**

**return：True(撞墙) / False(不撞墙)**

**draw\_maze()：画出当前的迷宫**

1. **编写自己的深度优先搜索算法def my\_search(maze):**

**首先以机器人起始位置建立根节点，并入栈；**

**接下来不断重复以下步骤直到判定条件:**

**·将栈顶位置标记已访问，判断栈顶是否为目标位置(出口)， 是则终止循环并记录回溯路径**

**·将栈顶节点出栈**

**·判断栈顶节点是否为叶子节点，是则拓展该叶子节点**

**·如果栈顶节点有子节点，则将每个子节点入栈**

**# 机器人移动方向和各种结果的奖励**

**move\_map = {**

**'u': (-1, 0), # up**

**'r': (0, +1), # right**

**'d': (+1, 0), # down**

**'l': (0, -1), # left**

**}**

**SearchTree：搜索树 add\_child：添加子节点；is\_leaf：判断是否为叶子节点**

**def expand(maze, is\_visit\_m, node):**

**"""**

**拓展叶子节点，即为当前的叶子节点添加执行合法动作后到达的子节点**

**:param maze: 迷宫对象**

**:param is\_visit\_m: 记录迷宫每个位置是否访问的矩阵**

**:param node: 待拓展的叶子节点**

**"""**

**def back\_propagation(node):**

**"""**

**回溯并记录节点路径**

**:param node: 待回溯节点**

**:return: 回溯路径**

**"""**

1. **编写QLearning算法：**

**·首先继承Robot类：**

**QRobot 类的核心成员方法**

**sense\_state()：获取当前机器人所处位置**

**return：机器人所处的位置坐标，如： (0, 0)**

**current\_state\_valid\_actions()：获取当前机器人可以合法移动的动作**

**return：由当前合法动作组成的列表，如： ['u','r']**

**train\_update()：以训练状态，根据 QLearning 算法策略执行动作**

**return：当前选择的动作，以及执行当前动作获得的回报, 如： 'u', -1**

**test\_update()：以测试状态，根据 QLearning 算法策略执行动作**

**return：当前选择的动作，以及执行当前动作获得的回报, 如：'u', -1**

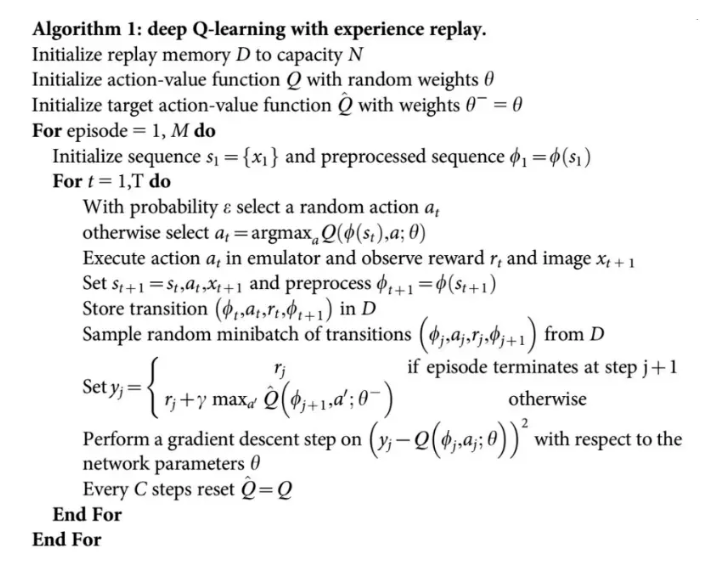
**reset()**

**return：重置机器人在迷宫中的位置**

**·QRobot 类实现了 QLearning 算法的 Q 值迭代和动作选择策略。在机器人自动走迷宫的训练过程中，需要不断的使用 QLearning 算法来迭代更新 Q 值表，以达到一个“最优”的状态，因此封装好了一个类 Runner 用于机器人的训练和可视化。**

**·主要编写train\_update()和test\_update()函数完成DQN算法**

**流程：**



1. **代码内容**

（能体现解题思路的主要代码，有多个文件或模块可用多个"===="隔开，必填）

====================================================================

基础搜索代码：

import numpy as np

# 机器人移动方向

move\_map = {

'u': (-1, 0), # up

'r': (0, +1), # right

'd': (+1, 0), # down

'l': (0, -1), # left

}

# 迷宫路径搜索树

class SearchTree(object):

def \_\_init\_\_(self, loc=(), action='', parent=None):

"""

初始化搜索树节点对象

:param loc: 新节点的机器人所处位置

:param action: 新节点的对应的移动方向

:param parent: 新节点的父辈节点

"""

self.loc = loc # 当前节点位置

self.to\_this\_action = action # 到达当前节点的动作

self.parent = parent # 当前节点的父节点

self.children = [] # 当前节点的子节点

def add\_child(self, child):

"""

添加子节点

:param child:待添加的子节点

"""

self.children.append(child)

def is\_leaf(self):

"""

判断当前节点是否是叶子节点

"""

return len(self.children) == 0

def expand(maze, is\_visit\_m, node):

"""

拓展叶子节点，即为当前的叶子节点添加执行合法动作后到达的子节点

:param maze: 迷宫对象

:param is\_visit\_m: 记录迷宫每个位置是否访问的矩阵

:param node: 待拓展的叶子节点

"""

can\_move = maze.can\_move\_actions(node.loc)

for a in can\_move:

new\_loc = tuple(node.loc[i] + move\_map[a][i] for i in range(2))

if not is\_visit\_m[new\_loc]:

child = SearchTree(loc=new\_loc, action=a, parent=node)

node.add\_child(child)

def back\_propagation(node):

"""

回溯并记录节点路径

:param node: 待回溯节点

:return: 回溯路径

"""

path = []

while node.parent is not None:

path.insert(0, node.to\_this\_action)

node = node.parent

return path

def my\_search(maze):

"""

任选深度优先搜索算法、最佳优先搜索（A\*)算法实现其中一种

:param maze: 迷宫对象

:return :到达目标点的路径 如：["u","u","r",...]

"""

path = [] # 记录路径

start = maze.sense\_robot()

root = SearchTree(loc=start)

stack = [root] # 节点栈，用于层次遍历

h, w, \_ = maze.maze\_data.shape

is\_visit\_m = np.zeros((h, w), dtype=np.int) # 标记迷宫的各个位置是否被访问过

while True:

current\_node = stack[-1]

is\_visit\_m[current\_node.loc] = 1 # 标记当前节点位置已访问

#出栈

stack.pop()

if current\_node.loc == maze.destination: # 到达目标点

path = back\_propagation(current\_node)

break

if current\_node.is\_leaf():

expand(maze, is\_visit\_m, current\_node)

# 入栈

for child in current\_node.children:

stack.append(child)

return path

====================================================================

DQN算法：

import random

from Maze import Maze

from Runner import Runner

from QRobot import QRobot

class Robot(QRobot):

valid\_action = ['u', 'r', 'd', 'l']

def \_\_init\_\_(self, maze, alpha=0.5, gamma=0.9, epsilon=0.5):

"""

初始化 Robot 类

:param maze:迷宫对象

"""

self.maze = maze

self.state = None

self.action = None

self.alpha = alpha

self.gamma = gamma

self.epsilon = epsilon # 动作随机选择概率

self.q\_table = {}

self.maze.reset\_robot() # 重置机器人状态

self.state = self.maze.sense\_robot() # state为机器人当前状态

if self.state not in self.q\_table: # 如果当前状态不存在，则为 Q 表添加新列

self.q\_table[self.state] = {a: 0.0 for a in self.valid\_action}

def train\_update(self):

"""

以训练状态选择动作，并更新相关参数

:return :action, reward 如："u", -1

"""

self.state = self.maze.sense\_robot() # 获取机器人当初所处迷宫位置

# 检索Q表，如果当前状态不存在则添加进入Q表

if self.state not in self.q\_table:

self.q\_table[self.state] = {a: 0.0 for a in self.valid\_action}

action = random.choice(self.valid\_action) if random.random() < self.epsilon else max(self.q\_table[self.state], key=self.q\_table[self.state].get) # action为机器人选择的动作

reward = self.maze.move\_robot(action) # 以给定的方向移动机器人,reward为迷宫返回的奖励值

next\_state = self.maze.sense\_robot() # 获取机器人执行指令后所处的位置

# 检索Q表，如果当前的next\_state不存在则添加进入Q表

if next\_state not in self.q\_table:

self.q\_table[next\_state] = {a: 0.0 for a in self.valid\_action}

# 更新 Q 值表

current\_r = self.q\_table[self.state][action]

update\_r = reward + self.gamma \* float(max(self.q\_table[next\_state].values()))

self.q\_table[self.state][action] = self.alpha \* self.q\_table[self.state][action] +(1 - self.alpha) \* (update\_r - current\_r)

self.epsilon \*= 0.5 # 衰减随机选择动作的可能性

return action, reward

def test\_update(self):

"""

以测试状态选择动作，并更新相关参数

:return :action, reward 如："u", -1

"""

self.state = self.maze.sense\_robot() # 获取机器人现在所处迷宫位置

# 检索Q表，如果当前状态不存在则添加进入Q表

if self.state not in self.q\_table:

self.q\_table[self.state] = {a: 0.0 for a in self.valid\_action}

action = max(self.q\_table[self.state],key=self.q\_table[self.state].get) # 选择动作

reward = self.maze.move\_robot(action) # 以给定的方向移动机器人

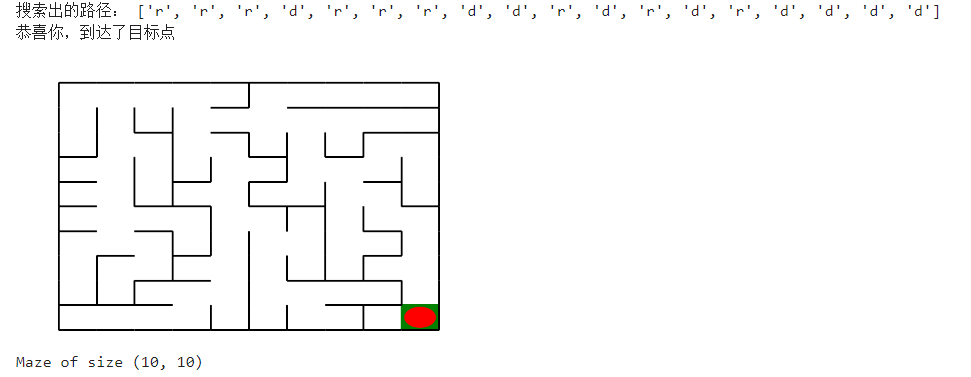
return action, reward

1. **实验结果**

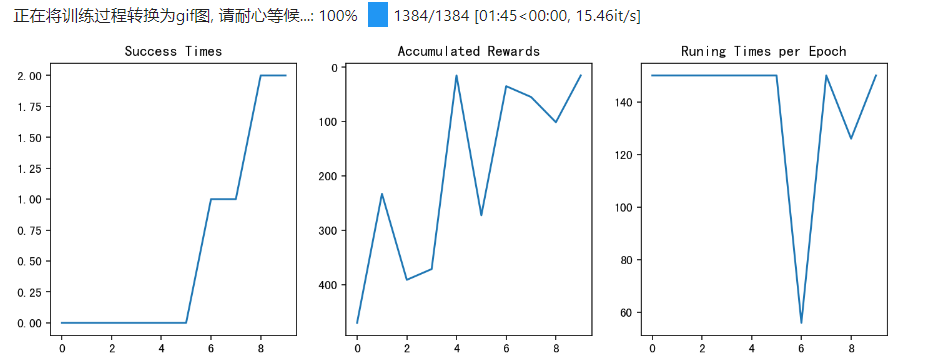
（实验结果，必填）

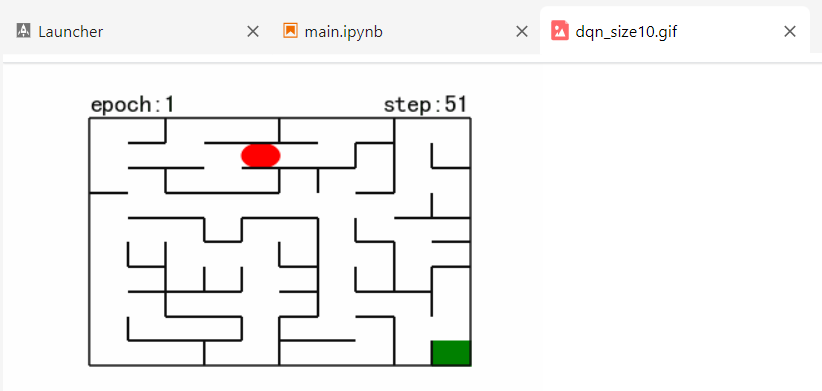
====================================================================

深度优先搜索算法结果：



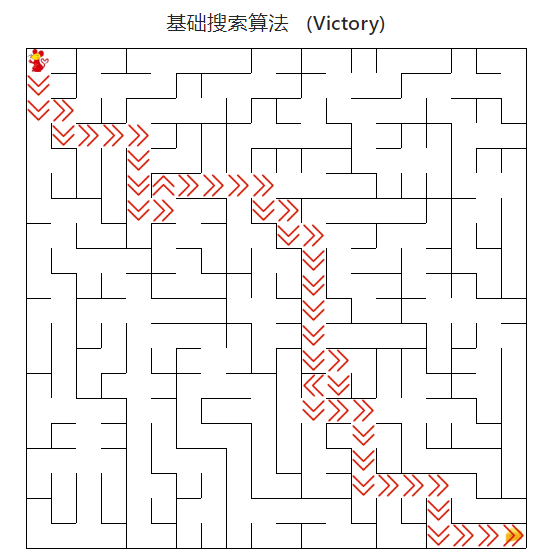
DQL算法结果（size=10）：



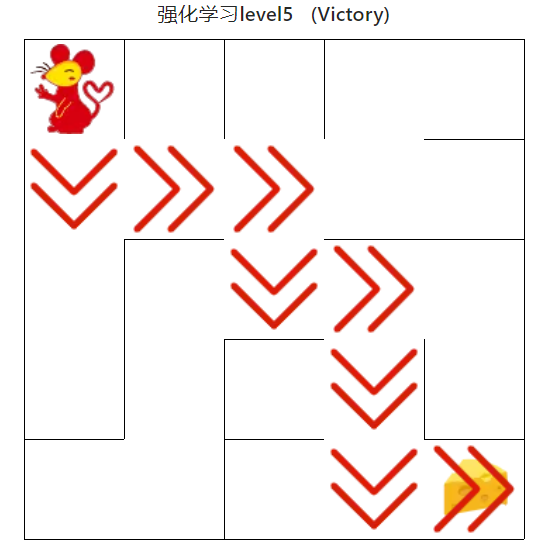


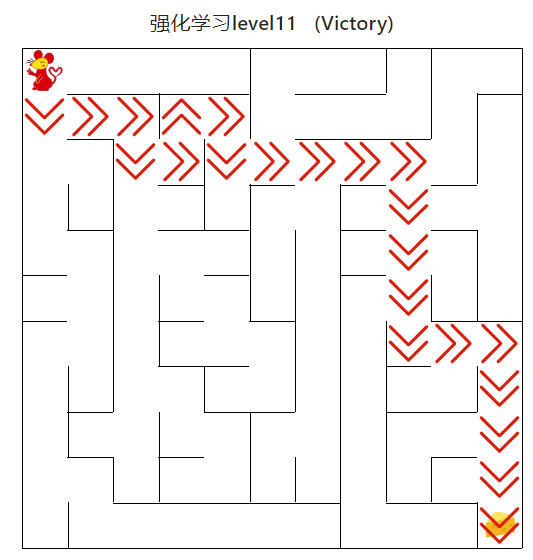
提交测试结果：











1. **总结**

（自评分析（是否达到目标预期，可能改进的方向，实现过程中遇到的困难，从哪些方面可以提升性能，模型的超参数和框架搜索是否合理等），**思考题，非必填**）

====================================================================

根据测试结果可知，达到目标预期。

可能不足的地方：本实验只尝试了size=10的迷宫，未尝试更大的，不确定能不能适用于更高规模的迷宫，而且执行过程中cpu负荷很重，差点爆掉，所以在性能方面还有待提高。

可以更改随机选择动作的可能性来进一步优化探索和利用之间的平衡，使得能够更快达到训练目的；也可以尝试用keras框架去实现，本实验用的PyTorch。

困难：对QLearning算法理解不足，较为抽象，经过一系列学习，得到预期结果。

强化学习是一个反复迭代的过程，每一次迭代要解决两个问题：给定一个策略求值函数，和根据值函数来更新策略。而 DQN 算法使用神经网络来近似值函数。