

# 程序报告

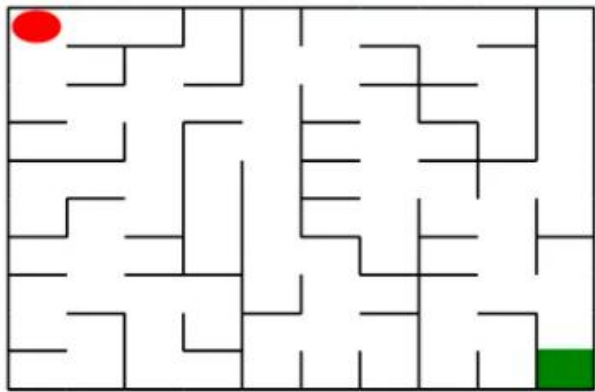
学号：2212046

姓名：王昱

## 一、问题重述

### 1.1 实验背景：

在本实验中，要求分别使用基础搜索算法和 Deep QLearning 算法，完成机器人自动走迷宫。



在任一位置可执行动作包括：向上走 'u'、向右走 'r'、向下走 'd'、向左走 'l'。执行不同的动作后，根据不同的情况会获得不同的奖励，具体而言，有以下几种情况：

- 撞墙
- 走到出口
- 其余情况

需要分别实现基于基础搜索算法和 Deep QLearning 算法的机器人，使机器人自动走到迷宫的出口。

### 1.2 实验要求：

- 使用 Python 语言。
- 使用基础搜索算法完成机器人走迷宫。
- 使用 Deep QLearning 算法完成机器人走迷宫。
- 算法部分需要自己实现，不能使用现成的包、工具或者接口。

### 1.3 对问题的理解：

机器人走迷宫问题涉及运用搜索算法和强化学习算法来导航和解决迷宫。搜索算法如 DFS 和 BFS 可以有效地找到迷宫的解，而强化学习中的 Q-Learning 算法通过值迭代来估计状态-动作对的值，从而最大化长期奖励。Q-Learning 通过持续更新和学习，使机器人在复杂环境中能够自适应地找到最优路径，实现智能决策。关键在于通过不断试错和学习，优化机器人在迷宫中的行为策略，提高解决效率和成功率。

## 二、设计思想

### 2.1 基础搜索算法

基础搜索路径方法主要有两种：

- 广度优先搜索（BFS）

广度优先搜索（BFS）是一种用于图或树的搜索算法，它按层次逐层展开搜索节点。具体步骤如下：

（1）初始化：以机器人的起始位置建立根节点，并将其入队。

（2）搜索过程：不断重复以下步骤，直到满足判定条件：

- 将队首节点的位置标记为已访问。
- 判断队首节点是否为目标位置（出口）。如果是，则终止循环并记录回溯路径。
- 判断队首节点是否为叶子节点。如果是，则拓展该叶子节点。
- 如果队首节点有子节点，则将每个子节点插入队尾。
- 将队首节点出队。

#### ● 深度优先搜索（DFS）

深度优先搜索（DFS）是一种用于图或树的搜索算法，它沿着树的每个分支尽可能深入，直到找到目标或没有未检验的子节点为止。具体步骤如下：

（1）初始化：将根节点放入栈中。

（2）搜索过程：

- 从栈中取出第一个节点，并检验它是否为目标。如果找到目标，则结束搜寻并回传结果；否则，将它的某一个尚未检验过的直接子节点加入栈中。
- 重复上述步骤，如果不存在未检测过的直接子节点，将上一级节点加入栈中。
- 重复上述步骤，直到栈为空。
- 若栈为空，表示整张图都检查过了，即图中没有欲搜寻的目标。结束搜寻并回传“找不到目标”。

这两种搜索方法在迷宫问题中都具有各自的优势，可以根据具体情况选择合适的方法，在本次实验中我使用了 DFS 深度优先搜索算法，代码如下：

```
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
```

#### 说明：

这段代码实现了一个深度优先搜索（DFS）算法，用于解决迷宫问题。其中，SearchTree 类，其中包含了添加孩子，判断是否是叶子结点等功能以及后面的两个函数 expand、back\_propagation 都是前面已经帮我们实现好了的。

首先，函数 my\_search(maze) 初始化起始位置，并创建一个根节点 root，然后将其压入栈中。通过一个二维数组 is\_visit\_m 来标记迷宫中的位置是否被访问过。在循环中，栈顶节点被取出并标记为已访问，然后检查是否到达目标位置（迷宫终点），若是则通过 back\_propagation 函数回溯路径并退出循环。如果当前节点是叶子节点，则调用 expand 函数生成子节点。将所有子节点依次压入栈中，继续搜索，直至找到目标位置。最终返回从起点到终点的路径。此方法通过栈结构实现 DFS，避免重复访问节点，提高了搜索效率。辅助函数 back\_propagation 用于路径回溯，expand 用于节点扩展。

## 2.2 Deep Qlearning 算法

该算法的主要流程如下图所示：

#### • DQN 算法流程

##### Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

For episode = 1,  $M$  do

Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

For  $t = 1, T$  do

With probability  $\epsilon$  select a random action  $a_t$

otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$

Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$

Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$

Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$

Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$

Every  $C$  steps reset  $\hat{Q} = Q$

End For

End For

- 继承 Robot 类 QRobot 类的核心成员方法
- 在我们的 Deep Qlearning 算法训练过程中首先读取机器人当前位置，之后将当前状态加入 Q 值表中，如果表中已经存在当前状态则不需重复添加，如下面的代码所示：

```
if self.state not in self.q_table:
```

```
    self.q_table[self.state] = {a: 0.0 for a in self.valid_action}
```

- 接下来这段代码描述了 Q-learning 算法中选择动作、执行动作和更新状态的过程。在训练过程中，机器人根据  $\epsilon$ -贪心策略选择动作（即有一定概率随机选择动作，或选择当前状态下 Q 值最高的动作），然后执行所选动作并获得奖励值 reward，最后获取执行动作后的新状态 next\_state。这些步骤帮助机器人逐步学习并优化在迷宫中的路径选择策略。

```

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)reward =
self.maze.move_robot(action)
next_state = self.maze.sense_robot()

```

- 接下来检查并更新 Q 值表:

```

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
return action, reward

```

- 在这段代码中，0.5 是用作  $\epsilon$ -贪心策略的衰减因子。每次更新后，将  $\epsilon$  乘以 0.5，使得探索概率逐渐减小，机器人更倾向于选择已知的最优动作。这种衰减策略平衡了探索新路径和利用已学得的路径，帮助机器人在初期多探索迷宫，后期更多依赖学得的最优路径，提高整体效率。衰减率过小，会导致随机性过强，影响性能；若衰减率过低，也不能很好完成训练，经过多次测试以及查阅资料，最后选择了最适合的 0.5，较为优秀地完成了本次实验。

### 三、代码内容

- 基础算法

```

# 导入相关包
import os
import random
import numpy as np
from Maze import Maze
from Runner import Runner
from QRobot import QRobot
from ReplayDataSet import ReplayDataSet
from torch_py.MinDQNRobot import MinDQNRobot as TorchRobot # PyTorch 版本
from keras_py.MinDQNRobot import MinDQNRobot as KerasRobot # Keras 版本
import matplotlib.pyplot as plt

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
```

- Q-learning:

```
import random
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() # 获取机器人当前所处迷宫位置
        # 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}

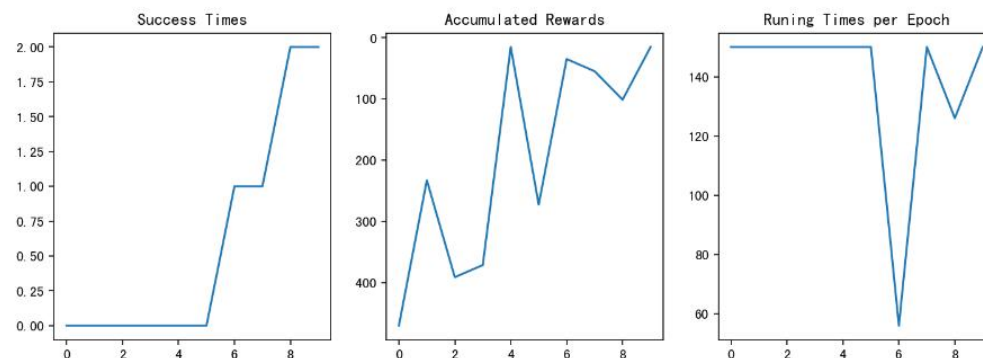
    action = max(self.q_table[self.state], key=self.q_table[self.state].get) # 选择动作
    reward = self.maze.move_robot(action) # 以给定的方向移动机器人
    return action, reward

```

#### 四、实验结果

训练过程的 gif 图:

正在将训练过程转换为gif图, 请耐心等待...: 100% 1384/1384 [01:45<00:00, 15.46it/s]



平台检测结果:



接口测试

✔ 接口测试通过。

用例测试

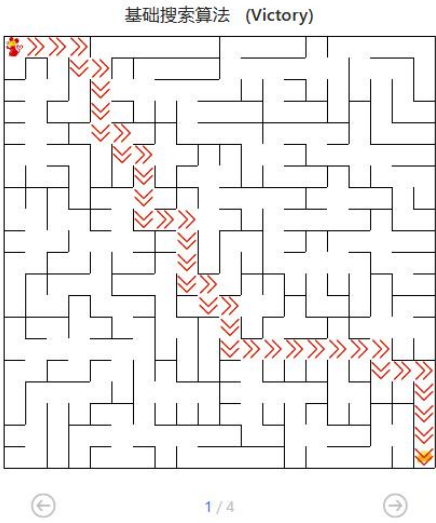
[展示迷宫](#) ▾

测试点	状态	时长	结果
测试基础搜索算法	✔	1s	恭喜, 完成了迷宫
测试强化学习算法 (初级)	✔	1s	恭喜, 完成了迷宫
测试强化学习算法 (中级)	✔	1s	恭喜, 完成了迷宫
测试强化学习算法 (高级)	✔	1s	恭喜, 完成了迷宫

样例点 1:






用例测试

[隐藏迷宫](#) ^



样例点 2:

隐藏迷宫 ^



隐藏迷宫 ^



隐藏迷宫 ^

A maze with a yellow cat at the start and a yellow arrow at the end. A red path of arrows shows the solution, starting from the cat and ending at the arrow.

→

## 五、总结

本次实验的各项指标都达到了预期水平，从基础搜索算法到我们 Q-Learning 实现都通过了系统的测试，表现良好。对于基础搜索算法部分，我们可以进一步优化，如使用 A\* 算法、Dijkstra 算法等来提高性能。对于第二部分的 Q-Learning 算法，我们可以尝试双向搜索算法等进行优化。此外，通过调整参数也可以实现性能优化，但这可能导致训练结果存在一定的不确定性。实验中遇到的最大问题是 Q-Learning 算法在找到路径后会停止，可能导致未找到最优路径，在复杂迷宫中可能绕远路或失败。

当代码遇到大规模的迷宫，对于大规模迷宫的解决方案，计算量可能会非常大，需要进一步优化算法和计算资源的利用。