hw3 机器人自动走迷宫

学号：22421023 姓名：杨宇轩

2.3 题目一 实现基础搜索算法

A*算法

A*算法是一种启发式搜索算法， 利用人为设计的启发函数先对一些更有希望的方向进行搜索。

本作业实现的是一种基于**宽度优先搜索**的A*算法，启发函数$h(x,y)=|x-des.x|+|y-des.y|$，即当前位置到终点的**曼哈顿距离**

关键代码如下：

```
class Node:
    def __init__(self, loc, des, action="", parent=None):
        self.loc = loc
        self.action = action
        self.parent = parent
        self.des = des

    def Hcost(self):
        x_des, y_des = self.des
        x, y = self.loc
        return abs(x_des - x) + abs(y_des - y)

    def __lt__(self, other):
        return self.Hcost() < other.Hcost()
```

定义了一个名为Node的类，内部实现了对于$h(x,y)$的计算，并实现了__lt__()方法以支持优先队列。

```
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",...]
    """
    h, w, _ = maze.maze_data.shape
    is_visit_m = np.zeros((h, w), dtype=np.int32)  # 标记迷宫的各个位置是否被访问过
    path = []
    q = PriorityQueue()
    start = maze.sense_robot()
    des = maze.destination
    root = Node(loc = start, des = des)
    q.put(root)
    while not q.empty():
        now = q.get()
        if now.loc == des:
            path = get_path(now)
            break
        x, y = now.loc
```

```
        if is_visit_m[x][y] != 0:
            continue
        is_visit_m[x][y] = 1
        can_move = maze.can_move_actions(now.loc)
        for a in can_move:
            new_loc = tuple(now.loc[i] + move_map[a][i] for i in range(2))
            new_node = Node(loc = new_loc, des = des,  action = a, parent = now)
            q.put(new_node)
    return path
```

A*算法从实现上类似Dijkstra算法，从一个优先队列中取出由$h(\cdot)$计算出的最优的候选方向，然后一步一步更新队列。

2.6 题目二 实现Deep QLearning算法

本题中最简单的QNetwork的实现基于MLP对当前坐标进行搜索，事实上，由于墙体的存在，处于$(1,1)$所选择的方向并不会与$(m-1,m-1)$所选择的方向有什么关系，因此可以考虑使用更加位置无关的计算方式。

```
class QNetwork(nn.Module, ABC):
    """Actor (Policy) Model."""

    def __init__(self, state_num: int, state_size: int, action_size: int, seed:
int):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
        """

        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.embedding_1 = nn.Parameter(torch.randn(state_num, 128, 4),
requires_grad=True)
        self.embedding_2 = nn.Parameter(torch.randn(state_num, 128, 4),
requires_grad=True)
        nn.init.xavier_normal_(self.embedding_1)
        nn.init.xavier_normal_(self.embedding_2)

    def forward(self, state):
        if len(state.shape) > 1:
            x = state[:, 0].long()
            y = state[:, 1].long()
        else:
            x = state[0].long()
            y = state[1].long()

        x_h = self.embedding_1[x.detach()]
        y_h = self.embedding_2[y.detach()]

        if len(state.shape) > 1:
```

```
            return torch.einsum("bdn, bdn -> bn", x_h, y_h)
        else:
            return torch.einsum("dn, dn -> n", x_h, y_h)
```

通过一种类似查表的方式更新参数，可以更加有效地更新DQN的参数。

这一方法可以通过所有的样例。