实验报告：迷宫寻路 AI

郑炜熹
软 72
2016013170

曾正
软 71
2017011438

ABSTRACT

本次实验报告介绍了 Markov 决策过程以及其描述，迷宫寻路 AI 程序的 python 实现以及实验结果.

Keywords

Markov Decision Process

1. 算法简介

Markov 决策过程有以下元素:

- 状态集合 S.
- 动作集合 A.
- 初始状态 s_0.
- 结束状态集合 $\{s_f\}$.
- 动作策略 π.

有以下函数:

- 后继概率函数 $P(s, a, s')$: 状态 s 执行动作 a 后转移到状态 s' 的概率.
- 回报函数 $R(s, a, s')$: 状态 s 执行动作 a 后转移到状态 s' 后获得的回报值.
- 行为价值函数 $Q_\pi(s, a)$, 表示在执行策略 π 时，对当前状态 s 执行某一具体行为 a 所能得到的回报值期望.
- 状态价值函数 $V_\pi(s)$, 表示某一状态在策略 π 下，从该状态开始的马尔可夫链收获的累计回报值期望.

该决策过程目的是为了求出最优策略 π^*:

$$\pi^* = \arg\max_\pi V^\pi(s), \forall s$$

因为对于任何 MDP, 总存在确定性的最优策略，则我们可以通过以下的迭代函数 (Bellman 方程):

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V_k(s')] \tag{1}$$

收敛时可以求出最优的状态价值函数 $V_*(s)$ 和行为价值函数 $Q_*(s, a)$, 之后我们便可以通过:

$$\pi^*(a|s) = \begin{cases} 1, & a = \arg\max_a Q_*(s, a) \\ 0, & otherwise \end{cases}$$

求出确定性的最优策略.

2. 地图设计和算法实例化

对于迷宫寻路 AI, 我们将地图方块分为 4 类:

- 普通方块.
- 障碍方块，当移动到该方块时会自动返回原位置.
- 陷阱方块，当移动到该方块时会返回起点.
- 终点方块.

动作分为四类: 上, 下, 左, 右. 每个方向都有分别 1/5 的概率移动到两个对角, 3/5 的概率移动到正方向.

根据这些特点, 我们可以设置对应的 $P(s, a, s')$ 和 $R(s, a, s')$:

- 当 s 通过 a 跳转的是普通方块或是终点方块, 则 s' 为对应的目标普通方块位置.
- 当 s 通过 a 跳转的是障碍方块, 则 s' 为 s 对应的位置.
- 当 s 通过 a 跳转的是陷阱方块, 则 s' 为起点位置.

概率 P 为对应的跳转概率值, 回报 R 根据目标方块设置回报值.

本次实验回报值设置为: 陷阱和终点分别设为负值和正值, 普通方块和障碍方块设为 0. 这种策略使得 AI 会尽量避开陷阱和障碍物, 寻找离终点最近的道路 (在 $\gamma < 1$ 情况下).

3. 算法实现

本算法采用 python 实现, 算法类为 Markov, 在 Markov.py 文件中.

该类有一些比较关键的成员:

- 变量 REWARD_<type>. 指定每种方块的回报值, 相当于回报函数.
- 变量 gamma. 衰减因子.
- 函数 read_map. 读取地图信息.
- 函数 initialize_state, 根据地图和回报值设定来初始化 P 和 R, 初始化算法的数据结构等.
- 函数 update_value_once, 执行一次 Bellman 方程迭代计算, 同时求出当前迭代下的最优策略.

算法实现的关键点:

- 数据结构的设计, 我们在 initialize_state 中设置了几个变量:
  - trans_dist 为一个字典, 字典的每一个 key 都指定了一个动作 (上下左右), 每个 key 指向一个二维列表, 其中第一维对应为前驱方块的位置, 二维为该前驱方框通过当前的 action(也就是 key 对应的动作) 能达到的方块的位置.
  - trans_prob 与上面类似, 只不过第二维为该前驱方框通过当前的 action 到达在 trans_dist 对应方块位置的概率.
  - reward 与上面类似, 第二维为该前驱方框通过当前的 action 到达在 trans_dist 对应方块后能获得的回报值.

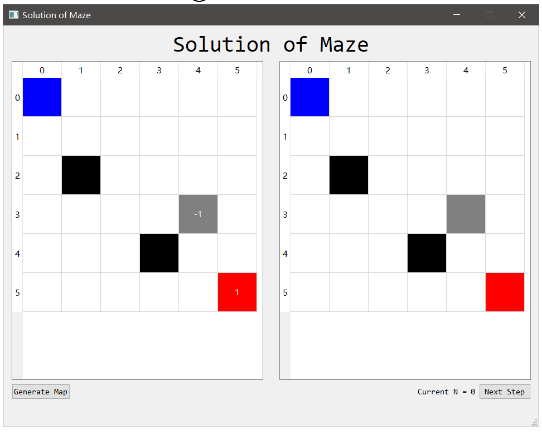
  这些数据结构构成了 $R(s, a, s')$ 和 $P(s, a, s')$. 相比于纯粹的矩阵, 在迭代时能节省一部分时间 (因为矩阵是稀疏矩阵, 第二维的大小为所有方格的总数, 而这个数据结构使得第二维能减小至有概率转移到的方块的个数). 执行函数后, 这些数据结构将会根据地图初始化.

- 算法的实现, 我们在 update_value_once 执行一次 Bellman 公式的迭代, 具体来说便是根据当前的价值函数 $V(s)$(初始为长度 $m*n$ 的零向量, m, n 分别为行列个数), 衰减因子 γ, $R(s, a, s')$ 和 $P(s, a, s')$ 来计算每个动作分别对应的价值, 然后求出最大值赋值于 $V(s)$ 中, 同时将对应的最优动作保存于每个方块中.

4. 实验结果展示

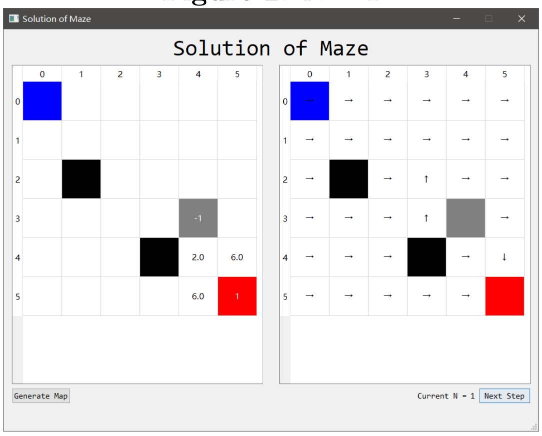
为了测试, 我们设计了以下的地图:

Figure 1: $N = 0$



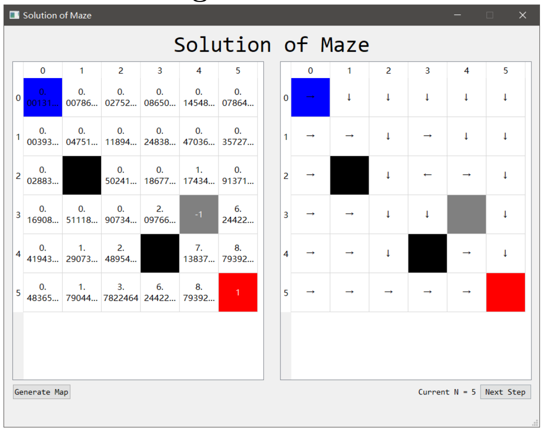
其中蓝色方块为起点, 红色方块为终点, 黑色方块为障碍物, 灰色方块为陷阱. 本次实验 $\gamma = 0.8$.

Figure 2: $N = 1$



在执行完一次迭代后, 可以看到终点周围的方块对应的价值根据迭代值更新了, 而对应右边表示的最优策略也指向了终点. 大部分没有更新的方块价值, 其对应的最优策略是 $\rightarrow$(默认值).

Figure 3: $N = 5$



而在执行了 5 次迭代后, 所有的方块的价值都被更新到非 0 值, 同时最优策略也被更新, 可以看到这些方向大都是朝着终点指向的.

为了获得收敛价值, 我们执行多次迭代:

Figure 4: $N = 30$

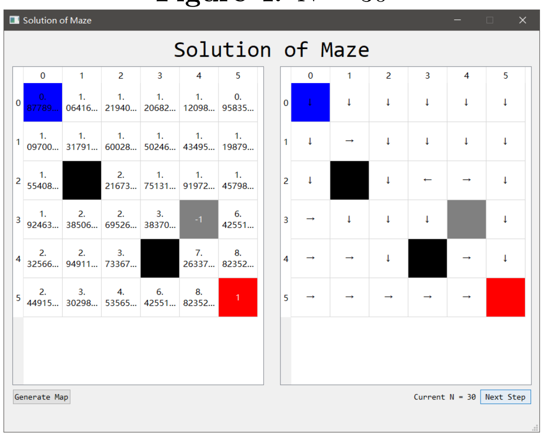
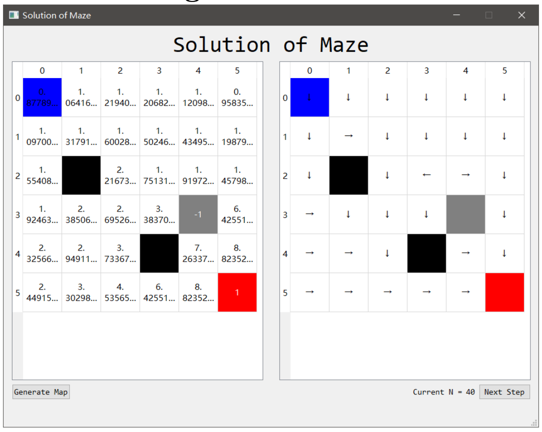


Figure 5: $N = 40$



可以看到, 执行到 $N = 30$ 时, 价值已经收敛, 这时候右边的策略便是最优策略.

5. 实验感想

本次实验实现的是一种比较简单的 MDP, 找到的最优策略为确定性策略, 而其他种类的 MDP 可能对应的是非确定性决策, 即 $\pi(a|s)$ 可能对应的是非 1 值. 对于这种情况, 算法的迭代和最优策略的选取还需要修改. 同时对于其他的 MDP, 一次迭代收敛得到的价值函数未必对应的就是最优的策略, 这时候应该通过当前得到的最优策略, 再次执行一次 Bellman 方程迭代过程, 然后再得出该步的最优策略, 重复, 直至得到收敛的最优策略.

MDP 作为强化学习的一种基本方式, 是进入机器学习大门的必备知识点, 通过这次大作业, 对 MDP 的了解会更加深入.

6. 附录

6.1 UI 使用说明

UI 采用 PyQt 库实现, 在运行时需要 pip 安装该库才能运行.

运行方式:

```
python App.py
```

Solution 界面:

- Label: Current N 为当前 N 的值
- Button: Generate Map 点击以打开窗口属性设置 Setting 界面
- Button: Next Step 点击以进行下一次更新，此操作会刷新两个地图界面，同时使 Label: Current N ++

Setting 界面:

- Input: Size x/y 输入整数以设置地图大小
- Input: Traps 输入一组点坐标作为陷阱，不同点坐标间以一个空格 (Space) 分隔
- Input: Barriers 输入一组点坐标作为障碍，不同点坐标间以一个空格 (Space) 分隔
- Input: Start 输入一个点坐标作为起点
- Input: End 输入一组点坐标作为终点，不同点坐标间以一个空格 (Space) 分隔
- Button: Random 点击以随机生成上述参数，可在随机的结果上再做修改
- Button: Generate 点击以按照上述参数生成地图