强化学习：第四次作业

程序源代码文件为`lovebird_markov.py`，程序的输出（环境界面及三种策略——仅策略评估、策略迭代、值迭代）放在`output`文件夹中。`.avi`文件是三种策略的寻路展示，由于在构建环境时我没有考虑计算量的问题，因此视频中的开头一段时间（静止状态）主要用于策略评估。

构建鸳鸯环境，体会MDP问题

由于上一次作业我只实现了鸳鸯环境，并没有将其进行MDP抽象。因此，在解决本次作业之前，我先完成鸳鸯环境的MDP抽象。

开发环境

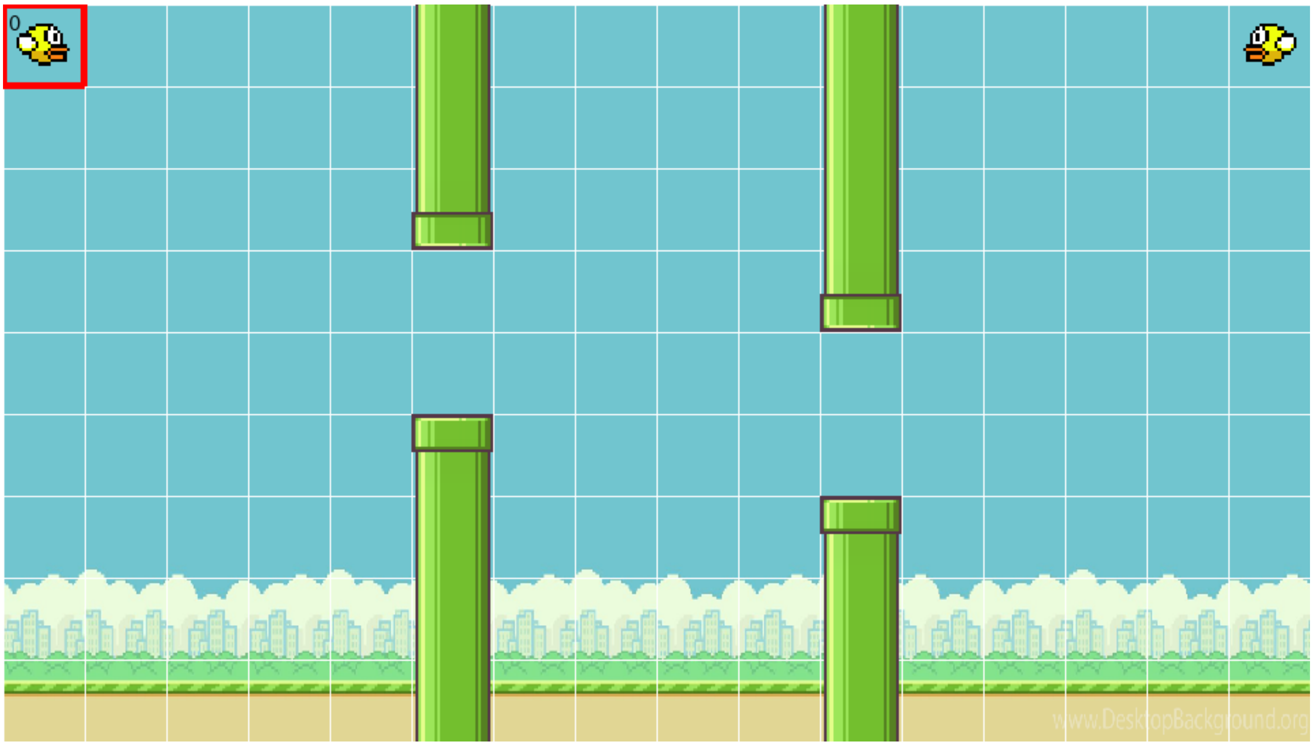
- ubuntu 18.04，i7-7700，16GB RAM，GeForce GTX 1080Ti
- python==3.6.6
- gym==0.10.8

鸳鸯环境

为了进行鸳鸯环境的MDP抽象，我重构了上次作业的`lovebirds.py`文件。`GameItem`类是环境中物品的基类，环境中的物品包括：网格`Grid`类、阻挡物`Brick`类、鸳鸯`Bird`类。鸳鸯环境的本体由`LoveBirdGame`类实现，环境的配置由`GameConfig`类进行管理。为了展示效果，游戏的帧率`fps=10`。

在`LoveBirdGame`类中，环境的逻辑控制依靠`loop()`函数实现，界面刷新由`blit()`函数实现。如果将`loop()`函数中第165行的代码注释解除，并注释掉167-169行的代码，那么`LoveBirdGame`就是上次作业的完成情况，即一个采用随机动作来寻找朋友的鸳鸯环境。

鸳鸯环境的初始界面如下图所示，环境界面的初始化由`game_env_init()`函数实现。界面中共有$16 * 9$个网格，左边鸳鸯的位置用红色的矩形框出，左上方的数字表示当前鸳鸯的移动步数。右边的鸳鸯位置固定。



MDP抽象

MDP抽象由`Markov`类实现，根据MDP问题的形式化表示，我将简要说明一下该类中变量的含义：

状态S

- `s_states`：左边鸳鸯可以到达的网格位置，即搜索（search）状态
- `c_states`：左边鸳鸯碰撞到阻挡物的网格位置，即碰撞（crash）状态
- `e_states`：左边鸳鸯与右边鸳鸯相遇的网格位置，即终止（end）状态

动作A

MDP问题的动作种类由`GameConfig`类中的`actions = ['e', 'w', 's', 'n']`控制。

状态转移概率P

在某个状态`s`和某个动作`a`的情况下，环境的状态转移概率由`get_pi_value(self, state, action)`函数实现。对于每个状态`s`和和每个动作`a`下的状态转移概率的值由`self.pi_values`变量（$16 * 9 * 4$维）存储，初始化的值为`0.25`。

立即回报R

每个状态`s`的立即回报由`self.i_rewards`变量（$16 * 9$维）存储，`s_states、c_states、e_states`三种状态的立即回报分别为`GameConfig.reward_category = [-1, -5, 1]`

折扣因子γ

累积回报的折扣因子由`GameConfig.gamma = 1`来决定。

策略评估——状态值函数$v_{\pi}(s)$

由于代码过长，请参见源程序中的`Markov.state_value_function(self, state)`函数，状态值函数的值存储在`self.v_values`变量中。阻挡物的状态值函数的值初始化为`-10000`（我也不知道这样做对不对），其他网格的状态值函数的值初始化为`0`。

策略评估——动作值函数$q_\pi(s,a)$

由于代码过长，请参见源程序中的`Markov.action_value_function(self, state, action)`函数，动作值函数的值存储在`self.q_values`变量中。

策略评估

由于代码过长，请参见源程序中的`Markov.policy_iteration(self)`函数。为了验证策略评估代码的正确性，我利用策略评估函数来计算课上PPT中的$4*4$网格问题，由`test_policy_evaluation()`函数实现，结果如下：

```
398 iterations for policy evaluation in random mode.
[[  0. -14. -20. -22.]
 [-14. -18. -20. -20.]
 [-20. -20. -18. -14.]
 [-22. -20. -14.   0.]]
```

可以看到，经过`398`次迭代以后，状态值函数$v_\pi(s)$的值保持稳定，输出值和PPT中的计算结果一致。

0.0	-14	-20	-22
-14	-18	-20	-20
-20	-20	-18	-14
-22	-20	-14	0.0

$$K = \infty$$

考虑到$4*4$的网格就需要`398`次迭代，因此设置了状态值函数值的精度：

```
last_v_values = np.around(last_v_values, decimals=0)
self.v_values = np.around(self.v_values, decimals=0)
```

这样，经过`24`次迭代就可以取得和原来`398`次迭代结果差不多的值。

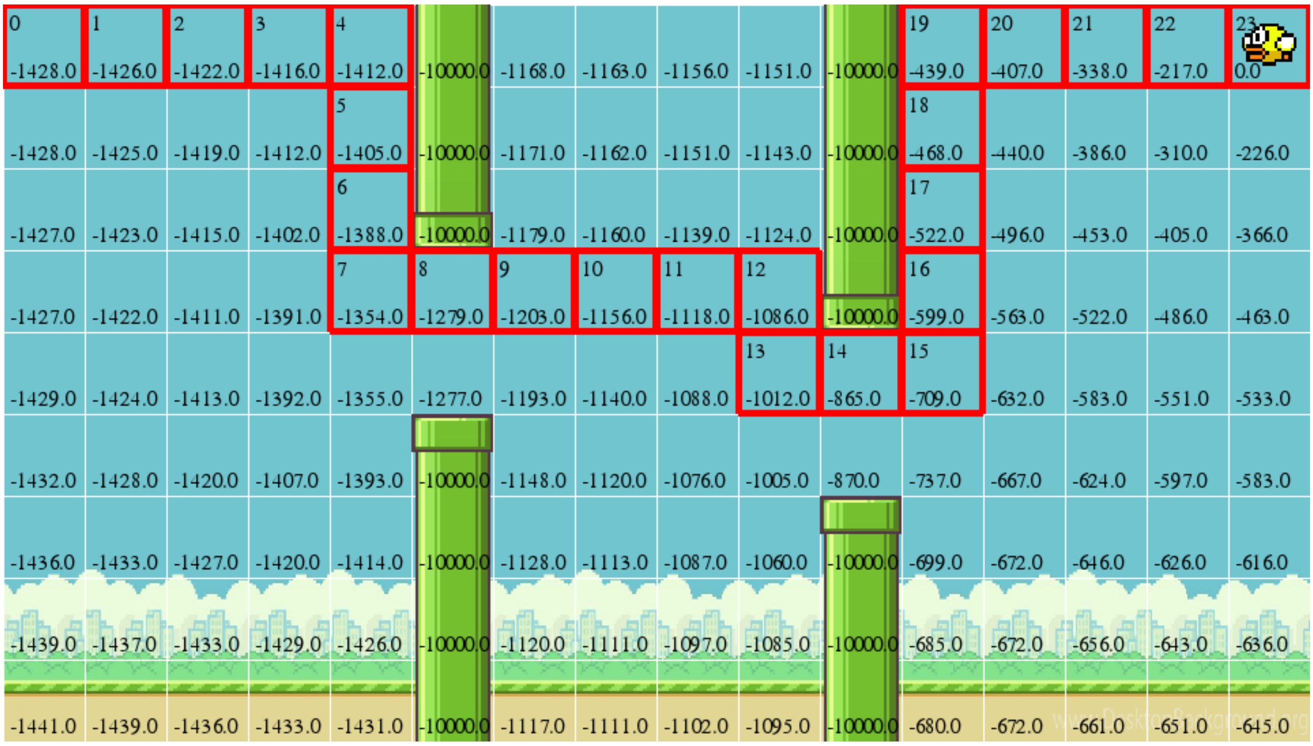
```
24 iterations for policy evaluation in random mode.
[[  0. -12. -17. -19.]
 [-12. -16. -17. -17.]
 [-17. -17. -16. -12.]
 [-19. -17. -12.   0.]]
```

动作选择

由于代码过长，请参见源程序中的 `Markov.give_action_advice(self, state)` 函数。根据状态值函数的值 `self.v_values`，依据贪心策略来选择当前状态下的动作。

随机动作的策略评估及其路径选择

在不进行策略改进的情况下，仅依靠策略评估进行决策的寻路结果如下图所示：



可以看到，经过 `23` 步以后，两只鸳鸯相遇。图中每个网格下方的数字表示经过策略评估后状态值函数的稳定值。

利用策略迭代和值迭代解决鸳鸯找朋友的问题

策略迭代

策略迭代通过 `Markov.policy_iteration(self)` 函数实现，由于需要在每次的迭代过程中进行策略评估，因此时间较长。实现时需要注意一下几点：

- 在每次迭代前，要将状态值函数的值 `self.v_values` 清空。

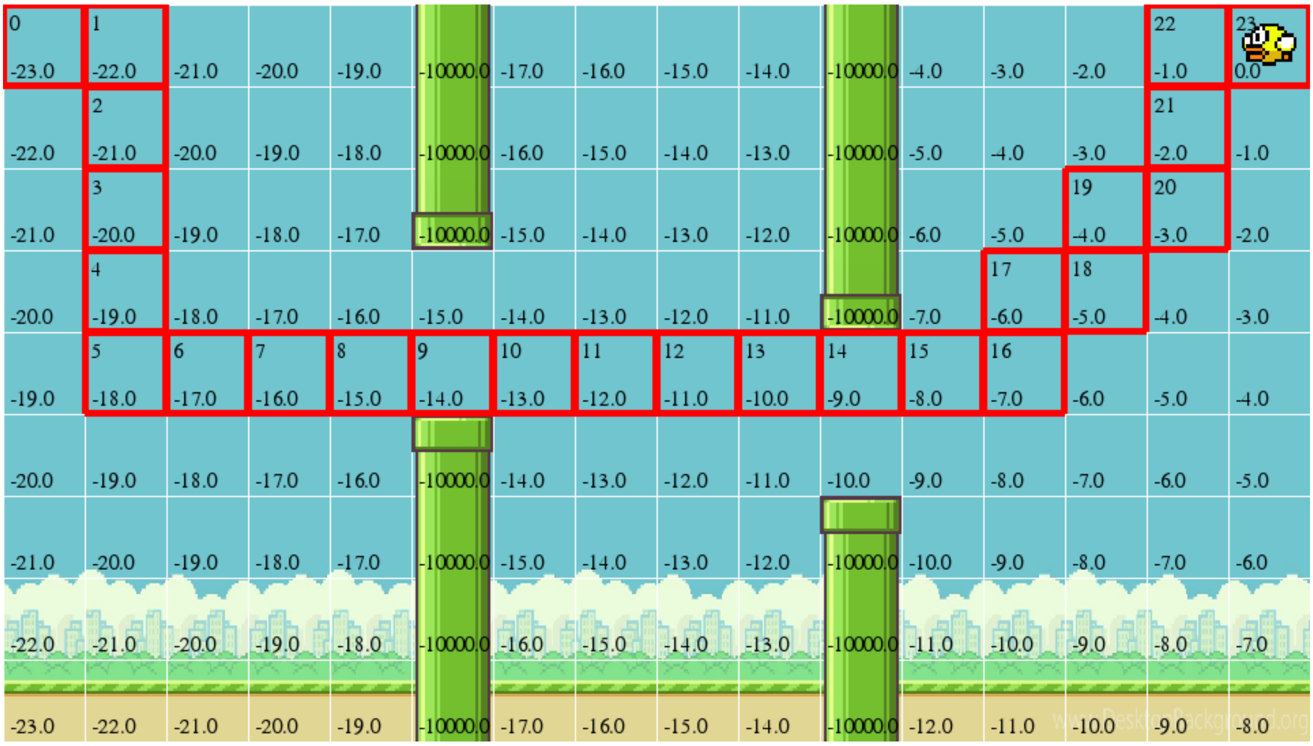
```
self.v_values = np.zeros_like(self.v_values, dtype=np.float)
for state in self.c_states:
    self.v_values[state[0]][state[1]] = -10000
```

- 更新策略的值`self.pi_values`的时候，需要注意可能存在多个极值的情况。因此，在出现多个极值的情况下，每个动作的状态转移概率是均等的。

  ```
  for i, state in enumerate(self.s_states):
      q_value_slice = self.q_values[state[0]][state[1]]
      max_q = max(q_value_slice)
      for j, q_value in enumerate(q_value_slice):
          if q_value == max_q:
              self.pi_values[state[0]][state[1]][j] = 1 / q_value_slice.count(max_q)
          else:
              self.pi_values[state[0]][state[1]][j] = 0
  ```

利用策略迭代的方式进行策略改进的寻路结果如下图所示：

```
1712 iterations for policy evaluation in policy_iteration mode.
20 iterations for policy evaluation in policy_iteration mode.
20 iterations for policy evaluation in policy_iteration mode.
3 iterations for policy iteration in policy_iteration mode.
```



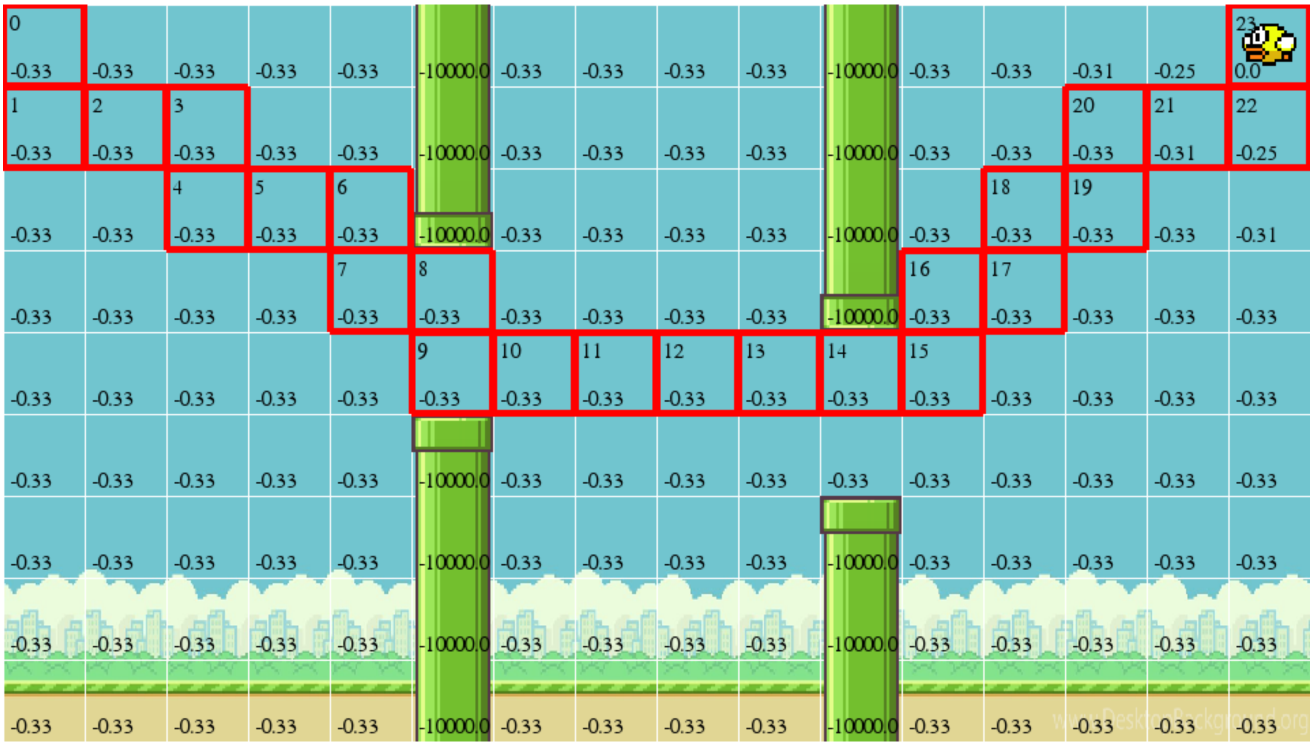
可以看到，策略迭代进行了`3`轮，每轮策略迭代中的策略评估的迭代次数分别是`1712、20、20`次。需要注意的是，判断策略迭代的终止条件`if (last_pi_values == self.pi_values).all()`并没有采用类似于策略评估时降低精度的方法。

值迭代

值迭代通过`Markov.value_iteration(self)`函数实现。每次迭代时，对于每个状态`s`，都考察该状态下每个动作`a`对应的动作值函数$q_\pi(s,a)$的最大值，并利用对应动作的状态值函数$v_\pi(s)$的值来更新该状态`s`的状态值函数$v_\pi(s)$的值。同样地，判断值迭代的终止条件`if (last_v_values == self.v_values).all()`也没有采用类似于策略评估时降低精度的方法。

利用值迭代的方式进行策略改进的寻路结果如下图所示：

```
24 iterations for value iteration in value_iteration mode.
```



可以看到，值迭代进行了`24`轮，但由于不涉及策略评估，因此计算时间远小于策略迭代的计算时间。由于网格的空间有限，因此在界面显示时，每个状态的状态值函数$v_\pi(s)$的值仅保留两位小数。而且我发现：相较于策略评估，值迭代的最有路径解的范围更大（请参见视频及图中状态值函数$v_\pi(s)$的值）。

总结

通过观察仅策略评估、策略迭代、值迭代三种寻路结果，可以发现：左边的鸳鸯通过走`23`步就能够找到右边的鸳鸯，三种方式的寻路结果所用的步数都相同，只是路径的选择不同。