人工智能导论作业 4：FreeWay 游戏

石睿（211300024、211300024@smail.nju.edu.cn）

（南京大学 人工智能学院，南京 210093）

摘 要： 基于 Free Way 游戏，理解强化学习中基于 ε -greedy 的 Q-learning 算法，并做出特征提取的优化和对强化学习中参数进行修正。

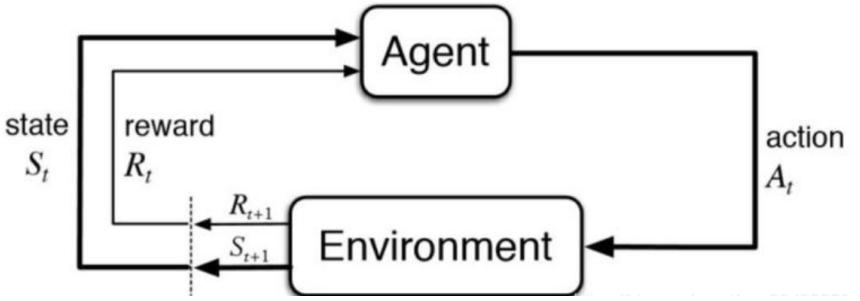
关键词： 强化学习； ε -greedy；Q-learning；

中图法分类号：TP301 文献标识码：A

1、 阅读代码，阐述强化学习的方法和过程

1.1 强化学习的一些通用方法和学习过程

强化学习的本质是让智能体通过尝试执行相应的决策来学习某个状态下的经验，其目标则是最大化累计价值。如下图所示，是一个 Agent（智能体）在进行决策并进行状态转移时的过程。当前状态为 S_t，Agent 根据现有的经验做出决策，这个决策可以是 exploit（依据现有经验选择当前状态下可以使累积价值最大的动作），也可以是 explore（随机选择一个未知的动作）。在做完这个动作 A_t 之后，转移到下一个状态 S_{t+1}，此时给予了 Agent 了一个单步反馈 R_t（可能是奖赏，也可能是惩罚）。



1.2 阅读代码，介绍本实验中强化学习的方法和过程

本实验中采用的 ε -greedy 的 Q-learning 算法是基于上述通用强化学习过程的一种离轨策略(off-policy)，在蒙特卡洛采样的基础上（代码中的 simulate 函数就是一个 MC sampling 过程）进行学习。之所以称之为离轨策略，因为其采样的策略和更新的策略完全不一样。在**采样**的时候，t 时刻只能前往某一个状态，然后获得这个状态给予的奖励。而在进行**决策评估**的时候，是要对 t 时刻所有可能的状态动作中挑选最大值（即下面 max 函数的作用）。下图之中即为离轨策略的 Q 值，此 Q(S,A)是从 S 状态执行动作 A 之后，获得累计价值的期望。

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R(S,A) + \gamma \max_a Q(S',a) - Q(S,A)]$$
$$= (1-\alpha)Q(S,A) + \alpha[R(S,A) + \gamma \max_a Q(S',a)]$$

代码具体过程为，main 函数连续对游戏进行 10 次采样训练，在每次训练的过程中调用 act 函数时通过 learnPolicy 来更新 Q 函数，完成强化学习并做出决策。

整个流程为 act->learnPolicy->simulate->weka.classifiers.REPTree，其中采样+训练学习的过程均在 learnPolicy 之中，最终完成对 Q 函数 m_policy（一个映射）的更新。

采样过程

下面 4 个步骤的最终目的是：**建立（s,a）—>Q(s,a)值的一个映射函数 f，并以此作为之后测试集的 Q 函数进行决策。**

Step1：首先调用 simulate 函数进行了 MC 采样，采样最大深度为 20 层，根据 policy（Q 值函数，也就是 ε -greedy）拿到了当前“最好”的动作。值得注意的是，此时 ε =0.3 的概率进行探索，有 0.7 的概率进行选择现有经验中可以最大化收益的动作。

Step2：其次，使用 heuristic 启发式函数对当前状态和走完拿到的“最好”动作之后的状态进行评估，分别计为 score_before 和 score_after。

Step3：根据打分结果把 state（提取的环境特征）、action（int 型变量，表示动作的编号）和 reawrd（前后两个分数之差乘以 factor,factor=0.99*0.99*0.99……其含义为程序不断运行的时候走一步的回报所占的比例会越来越小）保存在数组 sequence 之中。

Step4：依据 sequence 数组中的数据计算出 Q 值。依据 Q(s,a)的定义，从某一个状态执行某个动作之后，获得累积价值的期望。所以想要计算出 Q 值，只需要从当前状态开始算起，计算之后总的累积回报（reward）即可,也即代码中的过程

```
// calculate the acumulated Q
for (depth = depth - 1; depth >= 0; depth--) {
    accQ += sequence[depth].classValue();
    sequence[depth].setClassValue(accQ);
    data.add(sequence[depth]);
}
```

学习过程

最终依据此 4 个步骤拿到了的 Q 值函数（被返回到了 Instance 类型的 dataset 变量中），并保证这些拿到的 Q 值函数（状态、动作----Q 值的一组映射）不超过最大收集的状态数。

调用 m_policy.fitQ 函数，在 weka 之中利用 REPTree 模型完成训练，拟合出的 Q-learning 映射函数 f，并更新 Q 函数。

learnPolicy 做完之后，m_policy 也完成了更新，代码贪心的选择当前状态下可以得到最大 Q 值的动作进行返回，也即使用函数 getActionNoExplore。最后根据返回的动作对应的整形值，在 action_mapping 中依据此前建立的哈希函数找到最好的动作！

```
try {
    double[] features = RLDataExtractor.featureExtract(stateObs);
    int action_num = m_policy.getActionNoExplore(features); // no exploration
    bestAction = action_mapping.get(action_num);
} catch (Exception exc) {
    exc.printStackTrace();
}
```

Q1：策略模型用什么表示？该表示有何缺点？有何改进方法？

本强化学习的策略模型为 ε -greedy 的 Q-learning 算法，具体的应用方式如上文所述。其中 ε 是用于利用和探索之间的权衡，此时被取为 0.3。

缺点：ε 值是固定的，导致在整个学习过程中都只是以固定的概率进行探索和利用，这对于整个学习过程的前后期都不是非常合适的。

改进：因为游戏前期没有任何的先验知识，所以应该鼓励探索。到游戏后期基本所有可能都探索完毕，所以应该鼓励利用。综上，可以在不同时期设定不同大小的 ε 值，或者设定一个随时间或循环次数递减的 ε 。

Q2：Agent.java 之中 SIMULATION_DEPTH , m_gamma , m_maxPoolSize 三个变量分别有何作用？

SIMULATION_DEPTH 是 MC 采样过程中可以使用 ε -greedy 拿到每一次的“最好”动作并进行模拟运行的次数，设定为 20。

m_gamma 是 factor 的一个衰减因子，是贝尔曼方程中的系数 gamma，设定为 0.99。其作用为让程序在一次 MC 采样过程中不断运行时走一步的回报所占的比例越来越小。

贝尔曼方程呈形：$V^{\pi}(s) = E_{\pi}\left[R_t | s_t = s\right]$ ，其中 $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ ，r_{t+1} 表示从状态 s_t 转移到 s_{t+1} 的时候所得到的回报，也就是代码中 score_after、score_before 通过一个启发式函数拿到对当前局面的评价之后两者之差的值。gamma 是折损因子，也就是代码中的 m_gamma，取值为 0~1。

值得注意的是，在贝尔曼方程的递归展开形式进行计算的过程中，也就是上面公式中通过期望定义展开，会有一个概率 $P_{ss'}^{a}$ 表示在选择动作 a 时，状态由 s 转移到 s’ 的概率。在本游戏中当动作确定后，后面只有一种转移可能，这个时候 $P_{ss'}^{a} = 1$ 啦。

m_maxPoolSize 表示 m_dataset 中可以拿到的最大状态数量，设定为 1000。当从 MC 采样之中拿到的状态数量超过了 m_maxPoolSize，就会删除一部分状态，直到不超过最大状态数。

Q3：Qpolicy.java 代码中,getAction 和 getActionNoExplore 两个函数有何不同？分别用在何处？

getAction 函数中使用了 ε -greedy，有 ε 的概率随机选择一个动作来进行探索，有 1- ε 的概率选取当前已经收集的动作中的最优解。在 MC 采样的 simulate 函数之中使用，在采样的时候可以有几率探索到更多没有收集到的动作，有助于从局部最优解找到全局最优解。【采样中鼓励探索】

getActionNoExplore 函数直接选择当前已经收集的数据中最好的那一个。在 act 函数已经完成对 Q 函数的更新之后使用，在决策的时候依据 Q 值选择最好的动作呢！【决策时依据探索结果进行利用】

```
//在初始化的时候给了一个随机值，如果这个随机值比epsilon要小的话，就进行探索【从所有可选动作中随机选择一个！】
//下一个int值
// epsilon greedy
if( m_rnd.nextDouble() < m_epsilon ){
    bestaction = m_rnd.nextInt(m_numActions);
}
```

两个函数的代码差别

2 尝试修改特征提取方法，以得到更好的学习性能。说明修改的尝试和得到的结果

2.1 当前问题

当前代码中已经获取的信息有：以下 4 中物体的位置信息以及 4 个游戏状态。同时可以发现游戏中没有 npc，所以 getNPCPostions 几乎可以不使用。

```
//把当前状态（可以看做一张图）中的信息提取出来
LinkedList<Observation> allobj = new LinkedList<>();
if( obs.getImmovablePositions()!=null )//把所有不可动的物体的位置信息收集
    for(ArrayList<Observation> l : obs.getImmovablePositions()) allobj.addAll(l);
if( obs.getMovablePositions()!=null )//把所有可动的物体的位置信息收集
    for(ArrayList<Observation> l : obs.getMovablePositions()) allobj.addAll(l);
if( obs.getNPCPositions()!=null )//NPC的位置收集
    for(ArrayList<Observation> l : obs.getNPCPositions()) allobj.addAll(l);
```

```
//4个游戏状态信息
// 4 states
feature[868] = obs.getGameTick();
feature[869] = obs.getAvatarSpeed();
feature[870] = obs.getAvatarHealthPoints();
feature[871] = obs.getAvatarType();
```

观察游戏地图、游戏目标以及当前游戏情况。发现当前游戏有以下问题

• avatar 过于保守，在遇到了黄格子、多次遇到红格子导致掉血的时候，就不再会继续往上探索了。而是会保守的选择对当前最有利的走法：在右下角的位置一直左右徘徊（这样不会遇到黄格子和红格子了）

• avatar 不知道躲避黄格子和红格子，好像没有这一个概念一样。尤其是当 avatar 在中间的时候，它的上一行、同行、下一行可能都有黄、红格子，多次训练之后它仍然会一头撞上去。

2.2 可能的解决方案

综上两个问题，可以做出如下特征提取的改变：

- **问题 1：过于保守**

Potential option1：增加**当前 avatar 的位置、目标位置（最上面一行）、两者之间的距离**。此外，因为当前 avatar 一直在最后一行，他的 y 坐标和目标位置的纵坐标的差距一直是很大的。所以可以额外增加当前 avatar 距离**目标位置的纵坐标的差**。

```
Vector2d avatar_cur_pos=obs.getAvatarPosition();//【当前avatar的位置】

double dist_to_distination;//【距离目标位置的距离】
double y_to_distination;//【竖向到目标位置的距离】
```

```
/**
 * New observation. It is the observation of a sprite, recording its ID and position.
 * @param itype type of the sprite of this observation
 * @param id ID of the observation.
 * @param pos position of the sprite.
 * @param posReference reference to compare this position to others.
 * @param category category of this observation (NPC, static, resource, etc.)
 */
```

其中值得注意的是，可以通过 observation 类型变量中的 itype 属性来判断是不是目标位置呢！

Potential option2：optential option 1 在解决了不肯往上走的问题之后，为了避免一味地向上走或者一直在一个角落里边打转。因为尽管 avatar 在 act 函数之中可以通过 getAvailableActions 来拿到当前可以做的动作（会引起状态的变化），但仍然不可以避免多次循环的动作。所以可以**增加当前位置上、下、左、右距离当前 avatar 距离最近的阻隔物体信息**，让 avatar 提前预知做完当期动作之后可能遇到的阻碍物，来尽量少的来做出对状态不起到改变的一系列循环动作（如向上-向下-向上-向下……的循环）

```
double zhang_min_shang=0;//【当前距离avatar最近的上、下、左、右障碍物的距离是多少】
double zhang_min_xia=0;
double zhang_min_zuo=0;
double zhang_min_you=0;
```

- **问题 2：不知道躲避红、黄方块**

Potential option3：红、黄方块都是 movable 的物体，所以可以收集当前位置上一行、同行、下一行关于 movable 物体的信息，来让 avatar 学会躲避。

```
double ge_min_shang=0;//【当前距离avatar最近的上一行、同行、下一行最近的红黄格子距离是多少】
double ge_min_xia=0;
double ge_min_you=0;
double ge_min_zuo=0;
```

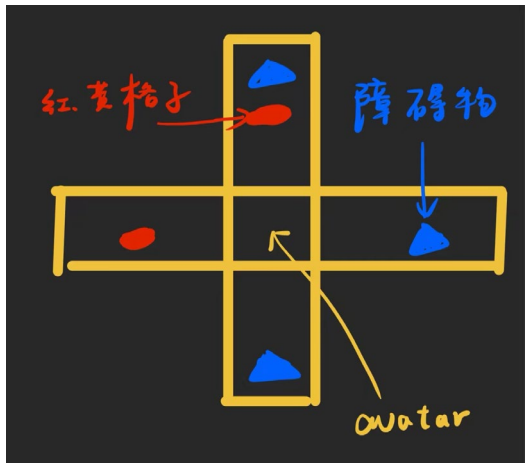
- **问题 3：无用的特征过多的被加入了**

Potential option4：删去无用的特征提取。当前收集的 800 多个特征之中有许多是在当前位置做出决策用不着的特征，比如说 NPC 的位置，离当前位置过远的可移动、不可移动的特征。加入这些对当前决策来说无用的特征会产生许多的干扰信息，

如删除 NPC 信息（已删除），以及只特征提取在 avatar 最近的 5*5 的方格之中的可移动、不可移动物体的信息（只是假象，没有实现）。

2.3 代码实现解决上述方案

以下为三个 potential option 的具体代码实现。要特别说明的是，当其中当涉及到要查找 avatar 上下左右障碍物或者红黄格子的时候以如下图所示的策略进行查找。因为当前策略动作可以立即做出，所以可以只考察当前 avatar 能做出的上下左右四个方向移动的路径上的信息即可。



Potential option1:

```
if(obs.getPortalsPositions()!=null)//目标位置信息收集【potential option1】
    for(ArrayList<Observation> l:obs.getPortalsPositions()) allobj.addAll(l);
```

```
//【potential option1】
for(Observation o:allobj){
    Vector2d dis=o.position;
    if(o.itype==4){
        dist_to_distination=avatar_cur_pos.dist(dis);//距离目标位置的距离
        y_to_distination=dis.y-avatar_cur_pos.y;//和目标位置纵坐标的差距
    }
}
```

Potential option2：

```
//【potential option2】
if( obs.getImmovablePositions()!=null ){//把所有不可动的物体的位置信息收集
    for(ArrayList<Observation> l : obs.getImmovablePositions()) {
        allobj.addAll(l);
        for(Observation o:l){
            if(avatar_cur_pos.y==o.position.y){//当前不可移动的障碍物和avatar在一条水平线上
                if(o.position.x<avatar_cur_pos.x)//左[没加28的时候]
                    if(zhang_min_zuo==0)
                        zhang_min_zuo=avatar_cur_pos.x-zhang_min_zuo;
                    else
                        zhang_min_zuo=Math.min(zhang_min_zuo,avatar_cur_pos.x-zhang_min_zuo);

                if(o.position.x>avatar_cur_pos.x)//右
                    if(zhang_min_you==0)
                        zhang_min_you=zhang_min_you-avatar_cur_pos.x;
                    else
                        zhang_min_you=Math.min(zhang_min_you,zhang_min_you-avatar_cur_pos.x);
                }
```

```
                if(avatar_cur_pos.x==o.position.x){//当前不可移动的障碍物和avatar在一条水平线上
                    if(o.position.y<avatar_cur_pos.y)//下
                        if(zhang_min_xia==0)
                            zhang_min_xia=avatar_cur_pos.y-zhang_min_xia;
                        else
                            zhang_min_xia=Math.min(zhang_min_xia,avatar_cur_pos.x-zhang_min_xia);
                    if(o.position.y>avatar_cur_pos.y)//上
                        if(zhang_min_shang==0)
                            zhang_min_shang=zhang_min_shang - avatar_cur_pos.y;
                        else
                            zhang_min_shang=Math.min(zhang_min_shang,zhang_min_shang-avatar_cur_pos.y);
                }
            }
        }
    }
```

Potential option 3：

```
//【potential option 3】
if( obs.getMovablePositions()!=null ){//把所有可动的物体的位置信息收集
    for(ArrayList<Observation> l : obs.getMovablePositions()) {
        allobj.addAll(l);
        for(Observation o:l){
            if(avatar_cur_pos.y==o.position.y){//当前不可移动的障碍物和avatar在一条水平线上
                if(o.position.x<avatar_cur_pos.x)//左
                    if(ge_min_zuo==0)
                        ge_min_zuo=avatar_cur_pos.x-ge_min_zuo;
                    else
                        ge_min_zuo=Math.min(ge_min_zuo,avatar_cur_pos.x-ge_min_zuo);

                if(o.position.x>avatar_cur_pos.x)//右
                    if(ge_min_you==0)
                        ge_min_you=ge_min_you-avatar_cur_pos.x;
                    else
                        ge_min_you=Math.min(ge_min_you,ge_min_you-avatar_cur_pos.x);
            }
```

```
            if(avatar_cur_pos.x==o.position.x){//当前不可移动的障碍物和avatar在一条水平线上
                if(o.position.y<avatar_cur_pos.y)//下
                    if(ge_min_xia==0)
                        ge_min_xia=avatar_cur_pos.y-ge_min_xia;
                    else
                        ge_min_xia=Math.min(ge_min_xia,avatar_cur_pos.x-ge_min_xia);
                if(o.position.y>avatar_cur_pos.y)//上
                    if(ge_min_shang==0)
                        ge_min_shang=ge_min_shang - avatar_cur_pos.y;
                    else
                        ge_min_shang=Math.min(ge_min_shang,ge_min_shang-avatar_cur_pos.y);
            }
        }
    }

}
```

2.4 修改后结果

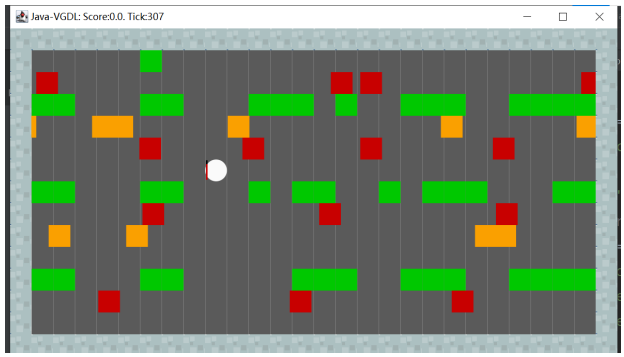
修改一带来的结果：增加了以上特征之后，avatar 不会只在右下角的位置来回移动了，而是会不断地向上尝试。在运行过程中发现还是会一直向右下角移动，所以又增加了和目标点之间的横坐标的差的绝对值。增加了之后，avatar 也会考虑在水平方向上和目标点的距离之差啦，也就不会一直向右下角移动了！

```
x_to_distination=Math.abs(dis.x-avatar_cur_pos.x);;//【竖向到目标位置的距离】
```

修改二带来的结果：增加了以上特征之后，avatar 不会一直执行重复的动作，而是根据上下左右的格子信息以及上下左右的障碍物信息让自己在遇到困境（死胡同）的时候尽量走出去！

修改三带来的结果：增加了以上特征之后，avatar 就会躲避红黄格子了，如果在上面或者下面一行，可

能会选择不动，或者会往格子离自己最近的那个方向的反方向移动。但由于红黄格子和 avatar 移动的速度不一样，可能有时候红黄格子会同时占据两个方块，但是 o.position 方法只会返回一个坐标，这样就可能会让 avatar 进行误判以为当前上下左右没有格子呢，进而掉血。



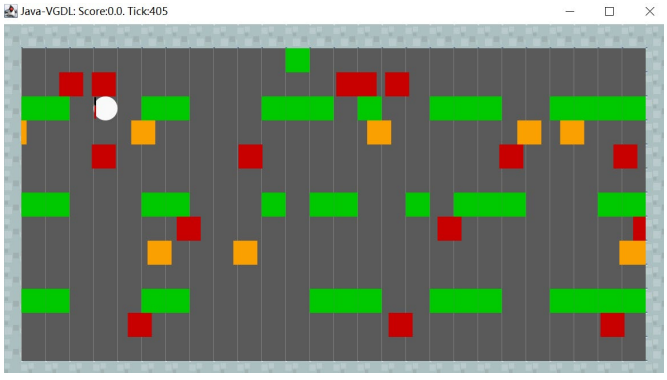
上图是实验中的最好结果

3 尝试修改特征提取参数，以得到更好的学习性能。说明修改的尝试和得到的结果

基于 ε -greedy 的 Q-learning 算法以及 simulate 中可供修改的参数有 SIMULATION_DEPTH，ε (算法中)，m_gamma（贝尔曼方程中）共三个参数。和模型无关的参数有程序训练的轮数（现在是 10），dataset 可以收集的最大特征数量 m_maxPoolSize。

对于 SIMULATION_DEPTH，由深度受限的深有优先搜索以及深度受限的蒙特卡洛搜索中，搜索深度应该是多多益善，但搜索深度的增加在另一方面肯定也会增加程序运行的时间。所以以下对 ε 和 m_gamma 的讨论之中把 SIMULATION_DEPTH 的大小定义成 50。

在不修改其他参数的前提下，运行程序，发现只增加 SIMULATION_DEPTH 可以更加鼓励 avatar 进行探索，而且运行了 20 次游戏发现此时可以提升平均水平到只修改特征提取时的最好水平！



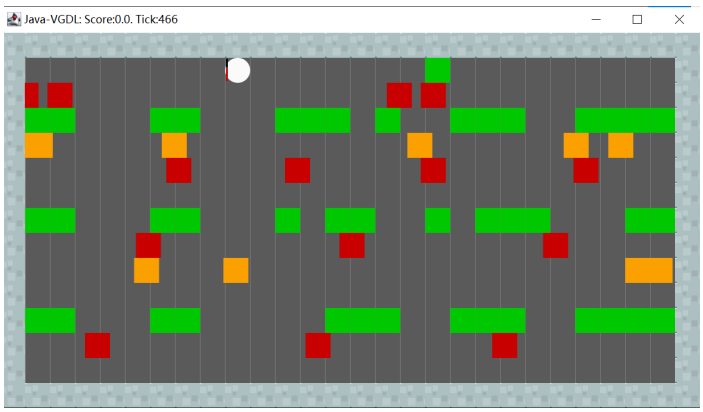
但可惜的是，由于血量的原因，以及在例如如图所示的位置上下都有频繁出现的红黄格子，在只增加搜索深度的情况下仍不能达到目标位置。

对于 ε，发现把 ε 略调大一点 **ε**，整个游戏都在不断进行探索，有时候还容易忽略红黄格子导致前功尽弃。因为每轮对 Q 值的更新会进行 10 次，所以此时我设定了一个随模拟次数降低的 epsilon 参数。注意到模拟的次数不是很多，所以在运行到最后的时候还是值得进行探索，同时又注意到了函数的下降速率。最终选择了函数 1-log(count/2+1)来对 epsilon 完成一个近似“随时间衰减”的变化。

```
//在初始化的时候给了一个随机值，如果这个随机值比epsilon要小的话，就进行探索【从所有可选动作中随机选择一个！】
//下一个int值
double temp=m_epsilon*(1-Math.Log10(count/2+1));
// epsilon greedy
if( m_rnd.nextDouble() < temp ){
    bestaction = m_rnd.nextInt(m_numActions);
}
```

对于 m_gamma，经过了多次尝试，把 m_gamma 从 0.99 调成 0.85 左右会获得更好的效果。因为对贝尔曼方程不是很熟悉，所以只能经过多轮尝试最后得到比较好用的 m_gamma=0.85。

最终在上述一系列的努力之下，avatar 终于可以到达和目标位置相同的层级了！



但此时成功率不高，大概能有 50%的概率到达目标位置。可能的原因可能是 heuristic 函数仅仅依据游戏的输赢来给出得分（SimpleStateHeuristic.java 中的 evaluateState 函数），对当前局面评估不是很全面。

对 heuristic 函数的修改其实和特征提取差不多，大概的思路如下：距离目标的距离、纵坐标的差值、最近的红黄格子以及障碍物。此外还需要把 avatar 的血量、最终 simulation-depth 后游戏的输赢都要考虑在内。最终多次尝试取得一个较好的比例把这些因素加起来得到对当前局面的评估！