利用A*算法实现Pacman游戏

201300096 杜兴豪

任务一：寻路算法

本任务中，仅设计了一个Agent和一个豆子，因此，本任务的问题等价于寻找从Agent到豆子的最短路径问题。

A^*算法步骤

> 1. 设置open_set, close_set为空集
> 2. 对于当前节点，生成子节点列表successors，并将当前节点放入close
> 3. 将全部子节点放入open，并计算启发式函数值
> 4. 对于open列表中的所有节点，找出启发值最小的节点用于下次探索
> 5. 当探索到目标节点后，停止拓展新节点

为了保证代码正确度，我首先实现了一种较为直观的启发式函数设计，来完成搜索框架。所选取的目标函数f满足：
$f(state) = h(state) + g(state)$
最开始设计的两种函数如下：

```
def myHeuristic(state, problem=None):
    start = problem.getStartState()
    g = util.manhattanDistance(start, state)
    h = util.manhattanDistance(problem.goal, state)
    return h + g
```

- h(x)：从当前位置到目标位置的曼哈顿距离manht(state, goal)
- g(x)：从起始位置到当前位置的曼哈顿距离manht(start, state)
- cost：每一步前进过程中的损失，本任务中恒定为1

优化过程

最开始时，我实现的A^*没有考虑到有多个相同分数点出现时的选取情况，默认为选择第一个

```
        best = min(open_set) # choose max f to explore
        open_set.remove(best)
```

但是这样就容易导致在一条非最优的路径上多次探索，降低其他路径被发现的概率，因此在代码中加入了随机选择片段如下

```
        best = min(open_set) # choose max f to explore
+++     choices = [choice for choice in open_set if choice[0] == best[0]]
+++     best = random.choice(choices)
        open_set.remove(best)
```

观察发现，在实现任务一的子问题2（smallMaze）的时候，若采用修改前的方案，拓展节点数一直保持在44个，而采用随机选取后，有时会出现42、43个，最差也是44个节点。

这样修改虽然可能在某些特定任务上看不出优势（有可能最优路径恰好一直在首位出现），但对于大量不同的任务而言，它从一定程度上能降低最坏情况出现的概率。

对整体架构的优化

在先前的讨论中，我将优先级f的两部分：h, g都放入了`myHeuristic`中，然而，由于传入参数的限制，我不能在该函数中表达路径过程。而函数中采用的，以起点到当前点的曼哈顿距离，作为损失函数，我认为有误：在smallMaze任务中，如果采用上述的启发值，则吃豆人将直接左拐，进入一段U型路线，对时间造成极大浪费。（也即：找到的路径并非最优路径）

此外，暂时完成任务一的设计后，尝试运行任务二的代码时发现，之前所编写的aStarSearch方法不具有问题迁移能力，无法正常运行任务二，可能是由于之前对state的了解不够所导致的。

针对上述问题，我对代码进行了两部分优化：

- 在`myHeuristic`方法中，取消对$g(state)$的定义
- 重构`aStarSearch`方法，使之具有记录当前点经过的路径的功能

重定义heuristic较为简单，在此不多赘述。

重构aStarSearch方法

函数的整体逻辑不变：

- 从open列表中pop出优先级最高（启发值最小）的节点，放入close中
- 对节点的successors进行遍历，如果不在close中，则计算它的启发值，放入open
- 重复上述步骤，直到找到goalState

我对两个列表中的元素进行了重新设计，新设计中，对每个节点在两个列表中的表示方法为：

- close：仅包含节点状态state，
- open：实现为一个三元组的列表，每个元组中包含:
  - f值：用于比较优先级
  - state：用于进行迭代
  - actions：记录走到当前节点所经历的所有action，便于计算cost

改变列表存储内容后，将优先值的计算过程放在search函数中进行，具体为

```
h = heuristic(suc_state, problem)
g = problem.getCostOfActions(suc_actions)
```

重构后运行发现，虽然探索节点数较之前稍有提升（以smallMaze为例，从45到55左右），但是路径是正确的（之前节点数少可能是探索不到位），并且，得分有了极大提升（481到491）。和nullHeuristic相比，节点数少了一半，证明修改有效。

注意：重构前的代码仍然在文件中，以注释形式存在。

与nullHeuristic对比结果：

对比结果为方便叙述，列出表格（表中的数据均为上下浮动不大的中间数据）：

maze种类	myHeuristic		nullHeuristic	
	拓展节点数	得分	拓展节点数	得分

maze种类	myHeuristic		nullHeuristic	
bigMaze	549	300	625	300
openMaze	617	456	1500	448
smallMaze	59	481	90	491

nullHeuristic为什么能跑？在最终实现的aStarSearch方法中，我们发现：代码循环停止的条件是：当前pacman1的状态是goal state。由于调用nullHeuristic时，所有节点被探索的可能性都是一样的，因此代码将不断从地图中探索节点，并且由于close集合的存在，节点是不会被重复探索的，因此第一问中唯一的豆子一定会被探索到，最终返回探索到豆子时的路径。

因此，可以认为nullHeuristic的结果是最差情况：agent不经过任何思索胡乱行走，直到找到目标。

观察到无论是得分还是拓展节点数，myHeuristic的结果要比null好很多，证明实现有意义。

任务二：最短路径

和任务一不同，这一问需要规划出一条吃掉所有豆子的最短路径。最终状态也是由豆子数量为0来定义的。在任务一中已经完成了aStarSearch的设计过程，在本问中将具体阐述启发式函数的设计过程。

处理nullHeuristic

我认为，在这一问中，相比于当前节点到终点的启发值而言，当前到起点的损失值要更为重要。我将损失值g定义为：从起点到当前位置所经过的路径下，每一步所需要的开销之和。为了任务一二的接口一致性，heuristic方法中无法传入有效信息。因此我将这一部分计算封装在aStarSearch方法中，具体表现为：

```
h = heuristic(suc_state, problem)
if str(heuristic)[:-20] != str(nullHeuristic)[:-20]:
        g = problem.getCostOfActions(suc_actions)
else:
        g = 0
```

将两问的接口进行了耦合，确保在调用nullheuristic时能得到$f = 0 + 0 = 0$。

foodHeuristic设计

一些尝试

在得出最终设计之前，我经过了多次尝试，对该问的接口和参数获得了良好的理解。

对代码组织有新理解的尝试

（此时还没有修改aStarSearch的组织，g值还是放在heuristic里面计算的）

后来我认为，为了让吃豆人走出最短路径，就一定要让它不要走重复，每一步都是确保正确的才行。

为了实现记忆吃豆人所经过的路径的功能，我利用了题目给出的heuristicInfo功能，记录每次探索的节点位置，下次再探索时，如果发现这个节点已经被探索过，就不继续探索了。

具体到实现，首先是heuristic设计的改动：

```
def foodHeuristic(state, problem):
    position, foodGrid = state
    if position in problem.heuristicInfo['state']:
        return 99999
    else:
        return foodGrid.count()
```

相应的，heuristicInfo键值对的初始化在aStarSearch中进行

```
def aStarSearch():
        ... # Initialization
        if not type(state[0]) == type(1) and len(problem.heuristicInfo.keys()) ==
0:
                problem.heuristicInfo['state'] = [] # for food
        ...# choice making progress
        if not type(state[0]) == type(1):# 判断是任务二
                problem.heuristicInfo['state'].append(state[0])
        ...
```

实现过程中，发现由于heuristic计算时，该点还没有被选取，因此所有点都在heuristicInfo中，最终所有节点的返回值趋同。方法有误。但是这样的思路促使我改进了代码组织，才有了任务一中描述的，将g值修改为路径，并且剥离出heuristic函数。

没能满足admissible的尝试

为了能更快地吃掉更多的豆子，我想到了一种方式：周围剩下豆子更多的情形要比周围没几个豆子的情形更容易更快完成任务。思考一种情景：

> 一共八颗豆子，观察吃豆人周围一圈：
> 可能1：周围能看到的就有八颗，转一圈则吃完
> 可能2：周围只有七颗，那吃这七颗豆子的同时，就要多观察周围的环境，探索出最后剩下的豆子的位置...

这是很显然的一种思路，实现起来也不复杂：

```
def foodHeuristic(state, problem):
    position, foodGrid = state
    for x in [-1, 0, 1]:
        for y in [-1, 0, 1]:
            if (not x == 0) and (not y == 0):
                if InGrid((position[0]+x, position[1]+y), foodGrid):
                    # InGrid是我设计的判断当前点是否还在棋盘内的方法
                    h += foodGrid[position[0]+x][position[1]+y] == False
                else:
                    h += 1
    return -h
```

在测试过程中，发现这种方式的效果奇差无比————达到nullHeuristic的将近两倍探索量，并且经常有非最优的情况发生。

分析思路：该函数在吃豆人周围空空如也的时候很难抉择，并且由于有计算路径的g值作为代价损失，吃豆人走到这个点后，发现周围所有的点都没有多好，只能胡乱选择一个硬着头皮往下扩展，探索很多没有意义的废点，导致探索量暴增。

并且从一个全满的点N，走到另一个完全不相交的全空的点N'，最短仅需移动3次，而

$h(N)=8-1=7, h(N')=0$(移动过程中吃掉了周围的一个)，而
$h(N')+cost(N,N')=0+3=3<7=h(N)$并不满足consistent的要求。更深一步研究，该方法下，目标点的h值完全是无法度量的，目标是吃完所有豆，因此目标处应该周围全空，
$h(target)=0$，然而，一个周围没豆子的点却不一定就是目标点，并不满足单调性，该方法甚至不admissible。只能是作为一种难以判别最终结果时，对当前状态的预测。

没能满足consistent的尝试

目标是吃掉所有豆，找出最短路径，我们可以对当前问题进行放松：只有一颗豆子。
那么这个问题就被转化成了任务一。heuristic的伪代码如下：

```
获得当前点坐标信息
在foodGrid中，计算距离最远的food坐标
返回当前点和最远food之间的Manhattan距离
```

但是根据NFL定理，天下没有免费的午餐，同时新增了一个难问题：选哪个豆子作为仅剩的豆子？我最开始的想法是：选择距离出发点最远的豆子。这个方法对只有一条路径的问题是比较好的，它可以很轻松的完成Search1。然而，在运行Search2时（最优路径是一个逆时针旋转90度的6）吃豆人首先走到了6的头部——最远的豆子处，然后再回头去吃那些剩余的豆子，但是此时它们的h值仍然是到6的头部，因此此时它们的heuristic值都是在增大的，吃豆人在“迎难而上”，只是因为它在一一遍历完这些没用的路径之后，发现了那个终点，因此在前进。这种方法在Search2的探索量巨大，可能就是这样的原因。

这个问题很好解决呀：当探索到之前设置的豆子之后，重新遍历foodGrid，找出下一个最远的豆子并记录，直到下一次访问到目标后再改变目标.....这下好了，在目标豆的位置不变的情况下，它一定是consistent的：当前点的$h(N)$表示距离那个目标豆的曼哈顿距离，到下一个点N'的开销为1，也是它们之间的曼哈顿距离，容易证明$cost(N,N')+h(N')\leq h(N)$，目标豆是最优的，并且满足$h(N^*)=0, h(N)\leq h^*(N)$，非常完美。

然而，这样的consistent，是"不consistent的"，它不连续。改变目标豆时，h值突变，成为新的极大值，然后开始下降，直到达到目标后，再突变增大.....

想要改进，也很容易：只要我们能确定最优路径上的最远豆子就可以啦！然而，在确定“最优路径上的最远豆”的过程中，我们可能就已经解决了最优路径的问题.....

最终采用的设计

在设计aStarSearch时，我加入了g值的计算：计算吃豆人到当前位置时所经过的路径长度。显然的，要得到consistent的启发式函数，我们要把h值和g值进行关联。对问题进行放松：

原问题的goal是吃完所有豆子，在吃豆子的路上有可能出现进行移动但未能获得豆子的情况，则可以将问题放松为：

> 假设每次移动都能吃到豆子，则最少需要移动多少次？

观察到问题设定的cost为1，每吃一个豆子需要移动一次，那么吃完所有豆子所需要的移动次数刚好等于此时的豆子数，因此有heursitic：

```
def foodHeuristic(state, problem):
        position, foodGrid = state
        return foodGrid.count()
```

容易看出，该函数满足：

1. $h(N)\leq h^*(N)$：问题被放松，可能有步骤吃不到豆。由这条知道该heuristic是admissible的
2. $h(N)\leq h(N')+cost(N,N')$：从N到N'路径上，只要出现了没吃到豆子的位置，则显然超过了N点的启发值

因此可以看出，该函数是consistent的。

运行结果

在给出的三种地图下，吃豆人的路径均为肉眼可分辨的最优路径。拓展节点数及分值和null的比较如下：

maze种类	**foodHeuristic**		**nullHeuristic**	
	拓展节点数	得分	拓展节点数	得分
search1	11	534	15	534
search2	97	614	750	611.6
search3	31	779	运行时间过长	-

更多尝试和经验

在设计出这样的heuristic之前，我还尝试过其他各种稍微更复杂一点的heuristic，比如上面介绍的几种，也比如：

1. 用当前位置到所有food的曼哈顿距离之和作为当前状态的启发值
2. 用当前位置周围food的个数作为启发值，越少分越高
3. 以最远豆子到当前点的距离作为H

（代码体现在foodHeuristic中的注释部分）

以及更多，但是这些heuristic效果都很平凡。虽然它们没能成功实现我所需要的功能，但是它们给予了我很多新的想法，比如以曼哈顿距离和为h值的方法，它提醒了我需要思考关于h和g的关系。虽然只看h来说，它们实际的功能不会相差太大，但是为了和g值一起实现一个更好的功能，它就必须要和"一条正确路径的cost"有一定的联系。如果采用当前位置到所有food的距离和来实现，虽然h值很有区分度，但是它一定程度上忽略了g值的作用——h过大，导致g值微小的改变在h来说没有意义。但是在最终的结果中，不可能有那么长的路径的。所以我才能把思路集中到找出能代表一条简单路径的启发值。

此外，这些尝试也告诉我奥卡姆剃刀原则的重要性。不要太复杂，简单的未必就不好。这些思路应用到以后的代码实践中，作用远远比一次简单的A*设计要大得多。