吃豆人程序的设计

《人工智能导论》实验报告

专 业 名 称： 计算机科学与技术

学 生 姓 名： 马玉坤

学 生 学 号： 1150310618

指 导 教 师： 李钦策

同 组 人 员： 李伟枫, 许浩禹, 张宁

二〇一八年一月

摘　　要

吃豆人项目是加州大学伯克利分校为了入门级人工智能课程 (CS188) 开发的。在本实验中，我们需要通过修改吃豆人程序的部分程序 (search.py, searchAgents.py)，从而扩充吃豆人小程序的功能，对各种搜索方法进行学习和实践。

关键词: 人工智能导论; 搜索算法; CS188; 吃豆人

目　录

1 简介

吃豆人项目是加州大学伯克利分校为入门级人工智能课程开发的项目。通过一系列的 AI 技术来玩吃豆人。但是吃豆人项目不专注于为视频游戏构建 AI，恰恰相反这些项目教授使用者 AI 基础概念，如知情的状态空间搜索，概率推理和强化学习。这些概念是真实世界的应用领域，如自然语言处理，计算机视觉和机器人。

这些项目允许学生可视化实现的技术的结果。它们还包含代码示例和明确的指示，但不要强迫学生趟过过多的脚手架。最后，吃豆人提供了一个具有挑战性的问题环境，需要创造性的解决方案; 真实世界的 AI 问题是具有挑战性的，吃豆人也是。

吃豆人系列项目一共分为五个小项目：Search, Multiagent Search, Reinforcement Learning, Ghostbusters 以及 Classification。其中的项目一——Search 就是我们本次的实验。在这个实验中，吃豆人将通过一系列构建的一系列的搜索算法实现达到特定位置并有效地吃到食物。包括深度优先搜索、广度优先搜索、一致代价搜索、A* 搜索、启发式搜索等 8 个问题。

这八个问题分别是：

1. 应用深度优先算法找到一个特定的位置的豆
2. 宽度优先算法
3. 代价一致算法
4. A* 算法
5. 找到所有的角落
6. 角落问题（启发式）
7. 吃掉所有的豆子
8. 次最优搜索

在这个实验的实现过程中需要编辑两个关键文件“search.py”和“searchAgents.py”，“search.py”中需要补充编写四个基本搜索算法，“searchAgents.py”中需要补充编写所有启发式算法的基本内容。除此之外还有三个比较重要的文件：“pacman.py”、“game.py”和“util.py”。“pacman.py”是游戏的主文件，描述了一个 Pacman GameState 类型。“game.py”说明了 Pacman 世界如何工作的逻辑并且描述了几种支持类型，如 AgentState，Agent，Direction 和 Grid。“util.py”是对我们来说非常重要的文件，包括了实现搜索算法可以用到的几种数据结构。

通过这个实验，我们可以更好的理解各种基本搜索算法，并且通过可视化的

方法可以更加有成就感，激发学习兴趣。在人工智能的领域继续研究下去。

2 算法介绍

2.1 编写通用搜索算法

2.1.1 待解决问题的解释

编写一个通用的搜索算法，来解决“应用深度优先算法找到一个特定的位置的豆”,“应用深度优先算法找到一个特定的位置的豆”,“应用深度优先算法找到一个特定的位置的豆”,“应用深度优先算法找到一个特定的位置的豆”等四个问题。

2.1.2 问题的形式化描述

输入：problem(吃豆人游戏搜索问题),open(遍历所用的数据结构),heurisitic(启发式算法)

输出：path[](由方向组成的数组，可以指示吃豆人从起始状态走到终止状态)

2.1.3 解决方案介绍

四种算法既有相似性也有相异性。相似性体现在四种算法的框架类似，相异性主要体现在两个方面。

第一个方面便是启发式函数的问题。在四个问题中，最复杂的问题是 A* 搜索问题，因为 A* 搜索问题需要启发式函数。换个角度来看，其他三种算法（深度优先搜索，广度优先搜索以及代价一致搜索）可以看做是启发式函数恒为 0。

第二个方面数据结构的不同。

因此四种问题只需分别定义好数据结构以及启发函数，调用通用搜索算法便可以获得解决方案。

2.1.4 所用方法的一般介绍

我们定义了一个函数 unifiedSearch（意为通用搜索），unifiedSearch 需要三个参数，首先是搜索问题，第二个是遍历所用的数据结构对象，最后一个是所用的启发式（估价）函数。

函数执行的第一步便是初始化过程，将初始状态的 f, g, h($f[state] = g[state] + h[state]$，其中 h 为启发式函数) 三个函数定义好，然后将初始状态 push 到数据结

构对象中。

接下来便是遍历过程，每次从数据结构对象中 pop 元素，设为当前的状态，然后判断当前状态是否为结束状态，如果是结束状态就返回找到的路径，否则就扩展新的状态，放到数据结构对象中。每次扩展子状态都需要使用启发式函数计算状态的 $f[childState]$ 函数值。

2.1.5 算法伪代码

算法 1 通用搜索算法

输入: $problem$ 搜索问题，$open$ 遍历所需的数据结构对象，$heuristic$ 启发式函数
输出: 指示从起点到终点的路径的方向列表

```
1: initialState ← problem.getStartstate()
2: 初始化 initialState 的 f, g, h 函数
3: open.push(initialState)
4: while open 非空 do
5:     state ← open.pop()
6:     将 state 加入 close 表
7:     if state 为结束状态 then
8:         return GETPATH(state, initialState)
9:     end if
10:    for childState in state 的子状态 do
11:        更新 childState 的信息
12:        将 childState 加入 open
13:    end for
14: end while
15: function GETPATH(dest, src)
16:    state ← dest
17:    path ← []
18:    while state ≠ src do
19:        path ← path + state 的转移方向
20:        state ← 父状态
21:    end while
22:    return path
23: end function
```

2.2 找到所有的角落

2.2.1 待解决问题的解释

需要重新定义问题状态，使状态能够表示四个角落的访问状态，使得该问题使用之前的搜索算法解决时，能够遍历所有的角落（左上、左下、右上、右下四个角落）。

2.2.2 问题的形式化描述

修改 searchAgents.py 文件中的 CornersProblem 类。编写 $\_\_init\_\_()$、$getStartState()$、$isGoalState()$ 以及 $getSuccessors()$ 方法，使得 CornersProblem 能够成为一个自洽的问题，且当问题解决时，四个角落都会被访问到。

2.2.3 解决方案介绍

其实，定义好这样一个问题，只需要考虑清楚如何将四个角落的访问状态加入到状态中。因此我们在状态设计中新增维数，添加表示角落访问状态的维度，从而来表示状态。

2.2.4 所用方法的一般介绍

在之前的问题中，状态的表示为一个二元组 (x, y)。在该问题中，我们设计了一个新的维度，用来存储四个角落的访问状态。新的状态形式为 $((x, y), corners)$，是一个新的二元组，该二元组的第一维是一个坐标二元组，第二维是一个 4 位二进制数 $corners$。

当 $corners$ 为 1111 时，表示四个角落都访问过，这也是终止状态的充分必要条件。当 $corners$ 为 1000 时，表示仅仅访问过左上角落。而当 $corners$ 为 0000 时，表示四个状态都没有访问过。初始状态即为 $(startingPosition, 0000)$。

2.2.5 算法伪代码

算法 2 角落问题

```
function getStartState(self)
    return (self.startingPosition, 0)
end function
function isGoalState(self, state)
    if state[1] = 15 then
        return True
    else
        return False
    end if
end function
function getSuccessors(self, state)
    successors ← []
    for action ∈ Directions do
        从 action 和当前位置获取新的位置
        从新位置和当前四个角落的访问情况获取新的四个角落的访问情况
        if 新位置可达 then
            successors ← successors+ 新状态
        end if
    end for
    return successors
end function
```

2.3 角落问题（启发式）

2.3.1 待解决问题的解释

编写上一个问题（角落问题）对应的 A* 算法的启发式函数。

2.3.2 问题的形式化描述

输入：当前状态 $state$，当前问题 $problem$

输出：当前状态对应的估价函数值

2.3.3 解决方案介绍

枚举访问顺序，通过曼哈顿距离测算最小的代价。

2.3.4 所用方法的一般介绍

由于没有访问的角落最多只有四个，所以我们可以枚举四个角落的访问顺序（最多 $4! = 24$ 种），找到代价最小的访问顺序，代价的计算是通过曼哈顿距离来测算的，这样效率比较高，拿到了满分。

2.3.5 算法伪代码

算法 3 角落问题启发式函数

输入： $state$ 当前状态，$problem$ 搜索问题

输出： 当前状态的启发式函数值

1: $minDist \leftarrow \infty$
2: **for** $perm \in [0, 1, 2, 3]$ 的四种排列 **do**
3: $dist \leftarrow$ 使用曼哈顿距离计算的按照排列顺序所需的最小代价
4: $minDist \leftarrow min(minDist, dist)$
5: **end for**
6: **return** $minDist$

2.4 吃掉所有的豆子

2.4.1 待解决问题的解释

编写 *searchAgents.py* 中的 $foodHeuristic$ 函数，能够成为使用 A* 算法解决 $FoodSearchProblem$ 的启发式函数。

2.4.2 问题的形式化描述

补充 *searchAgents.py* 中的 $foodHeuristic$ 函数，使得其是一个 consistent 且 admissible 的启发函数。

输入： 当前状态 $state$，当前吃掉所有豆子的问题 $problem$

输出： 一个数字，表示当前状态的估价函数值

2.4.3 解决方案介绍

在吃掉所有豆子的问题中，状态的定义为 $((x, y), foodList)$。$foodList$ 表示所有豆子的位置列表。当豆子列表为空时，显然估计函数值应该为 0。当豆子列表不为空的时候，我们只需要计算所有豆子中，与当前位置曼哈顿距离最大的豆子，距离当前位置的精确距离（使用 *searchAgents.py* 中的 $mazeDistance$）就能达到非常惊人的效果，获得了 5/4 的分数。

2.4.4 所用方法的一般介绍

当豆子列表为空时，显然估计函数值应该为 0。当豆子列表不为空的时候，我们只需要计算所有豆子中，与当前位置曼哈顿距离最大的豆子，距离当前位置的精确距离（使用 *searchAgents.py* 中的 $mazeDistance$）。

2.4.5 算法伪代码

算法 4 吃掉所有豆子的启发式函数

输入： $state$ 当前状态，$problem$ 搜索问题

输出： 当前状态的启发式函数值

1: $position \leftarrow$ 当前状态的位置
2: $foodList \leftarrow$ 当前状态中所有没吃掉的豆子的位置列表
3: **if** $|foodList| = 0$ **then**
4: **return** 0
5: **else**
6: $farestFood \leftarrow foodList$ 中距离 $position$ 最近的豆子的位置
7: **return** $farestFood$ 距离 $position$ 的精确位置 ($mazeDistance$)
8: **end if**

2.5 次最优搜索

2.5.1 待解决问题的解释

定义一个优先吃最近的豆子函数是提高搜索速度的一个好的办法。补充完成 searchAgents.py 文件中的 AnyFoodSearchProblem 目标测试函数，并完成 searchAgents.py 文件中的 ClosestDotSearchAgent 部分，在此 Agent 当中缺少一个关键的函数：找到最近豆子的函数。

2.5.2 问题的形式化描述

补充 *searchAgents.py* 中的 $ClosetDotSearchAgent$ 类中的 $findPathToClosetDot$ 函数。

输入： gameState 游戏状态

输出： 一条路径，能够从当前位置走到最近的豆子的位置

2.5.3 解决方案介绍

使用之前定义的 bfs 函数，来解决这个问题即可。

2.5.4 所用方法的一般介绍

使用 $search.bfs(problem)$ 可以调用之前写好的 bfs 函数，返回值即为当前位置到最近的豆子的位置的路径。

2.5.5 算法伪代码

算法 5 找到到最近的豆子的路径

输入: $gameState$ 游戏状态

输出: 从当前吃豆人的位置到最近的豆子的路径

1: $problem \leftarrow$ AnyFoodSearchProblem($gameState$)
2: $path \leftarrow$ bfs($problem$)
3: **return** $path$

3 算法实现

3.1 实验环境与问题规模

3.2 数据结构

3.3 实验结果

3.4 系统中间及最终输出结果

4 总结及讨论

参考文献

附录 A　测试

A.1　第一个测试