同济大学
TONGJI UNIVERSITY

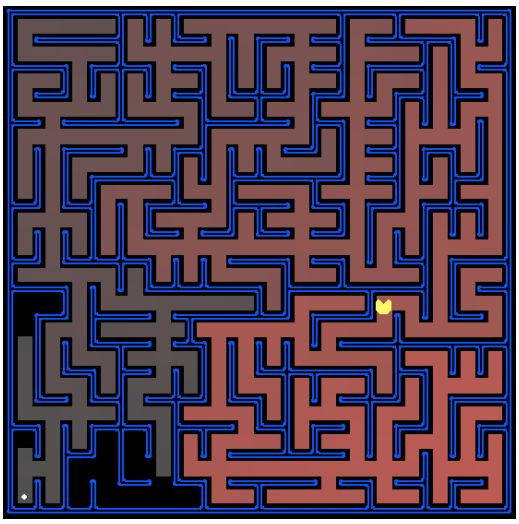
人工智能原理与技术
Project1实验报告

学　　号：2154312
姓　　名：郑博远
教　　师：王俊丽
完成日期：2023 年 4 月 3 日

1. 问题概述

1.1. 问题的直观描述

在如下图所示的迷宫世界中，白色圆点代表着食物，蓝色部分代表着墙壁，黄色的吃豆人智能体 Pacman 将会在其中寻找路径，其目标既可能是到达特定位置，也可能是高效地收集食物。Project1 将构建通用的搜索算法并将其应用于 Pacman 场景中。本次 Project 共分为 8 个子问题，具体描述如下：



问题 1 到问题 4 中，智能体 Pacman 通过 SearchAgent 逐步规划出路径并逐步执行，实现 Pacman 智能体从起始路径到固定位置的食物的寻路。

- 问题 1 中，通过实现深度优先搜索（Depth First Search, DFS）来构建 Pacman 智能体的路径规划；
- 问题 2 中，通过实现广度优先搜索（Breadth First Search, BFS）来构建 Pacman 智能体的路径规划；
- 问题 3 中，通过实现一致代价搜索（Uniform Cost Search, UCS）来构建 Pacman 智能体的路径规划；
- 问题 4 中，通过实现 A*搜索（A Star Search），并以曼哈顿距离作为作为启发式函数，来构建 Pacman 智能体的路径规划；

问题 5、问题 6 中，探究了智能体 Pacman 的新任务，即对于每个角落都有四个点的角落迷宫，找到 Pacman 从初始位置穿过迷宫并触及所有四个角落的最短路径（无论迷宫实际上是否有食物）。

- 问题 5 中，需要选择一种状态表示形式，以编码检测是否已到达所有四个角落所

需的所有信息；这样的抽象状态表示应该不编码无关信息（如幽灵的位置、额外食物的位置等）；

- 问题 6 中，需要寻找对于角落迷宫寻路较为合适的 A*算法启发式函数。启发式函数的返回值是一个对目标状态距离的近似值，当启发式函数的值恒为 0 时，A*算法实际上就是问题 3 中的 UCS 搜索算法。选取合适的启发式函数能够筛选掉无关的状态结点，实现更迅速的路径规划。本题中也以 A*算法的拓展结点数作为衡量启发式函数好坏的评价标准。

问题 7、问题 8 中，探究了智能体 Pacman 在迷宫中多个位置存在食物时，以尽可能少的步骤数进行的路径规划。

- 问题 7 中，需要为 A*算法定义合适的启发式函数来进行路径规划。与问题 6 中类似，当选择启发式函数值为 0（即使用 UCS 算法进行路径规划），扩展结点较多、寻路效率低下，因此需要设计合适的启发式函数。
- 问题 6 中，定义算法让 Agent 每次都贪婪地找寻最近的食物。

1.2. 项目已有代码的阅读与理解

1.2.1. `util.py`

该部分提供了搜索算法的有用数据结构，包括栈（LIFO，后进先出）、队列（FIFO，先进先出）与优先队列（保持队首元素最小），可分别用于深度优先搜索、广度优先搜索与 A*搜索（或一致代价搜索）中。其中 PriorityQueue 在向队列加入元素时额外传入一个值作为排序的一句；此外，还提供了继承 PriorityQueue 的 PriorityQueueWithFunction 类，可以通过通过传入 priority function 作为依据进行排序。

1.2.2. `pacman.py`

该文件运行 Pacman 游戏的主要文件。其中主要与外部交互的是 pacman.py 中定义的 GameState 类，其涉及到与智能体 Pacman 相关的信息，具体包含食物、智能体配置、分数变化等信息。在 searchAgent.py 文件中对具体问题 searchProblem 类的定义与初始化中，就包含了 GameState 类的信息。

GameState 类中常用的方法有：getPacmanPosition，能够获取到当下吃豆人智能体所在的位置；getFood 与 getNumFood，能够返回食物所在的位置与食物的数量；hasFood 与 hasWall，能够判断指定位置是否有食物或墙壁；isWin 和 isLose 能判断游戏进行情况。

```
class GameState:
    """
    A GameState specifies the full game state, including the food, capsules,
    agent configurations and score changes.

    Note that in classic Pacman, Pacman is always agent 0.
    """
```

1.2.3. game.py

该文件描述了 Pacman 游戏世界运作的逻辑。它包含了几种类的定义，包括 AgentState、Agent、Direction 和 Grid。

```
class AgentState:
    """
    AgentStates hold the state of an agent (configuration, speed, scared,
etc).
    """
```

game.py 中给出了 Agent 类的抽象类定义，规定了 Agent 类必须要具备 getAction 方法；具体的各类型 Agent 继承类在 searchAgent.py 中定义。AgentState 类用于存储智能体的当前状态。在本次 Project 中，AgentState 信息包括其配置、速度等。

```
class Configuration:
    """
    A Configuration holds the (x,y) coordinate of a character, along with its
    traveling direction.
    """
```

Directions 类主要用于东、西、南、北、左、右等方向常量的定义，使代码具有更高的可读性。Configuration 类用于存储角色的位置与运动方向。位置坐标系和常见的坐标系较不同，它以游戏窗口的左下角为原点，x 轴向右，y 轴向上，因此向上运动是沿着 y 轴的正方向，即(0,1)方向。其中的 generateSuccessors 方法在 Project1 中较常使用，用于获得当前位置 Agent 可行的后继位置。

Grid 类是封装好的二维列表。与 Configuration 类中规定相同，grid[x][y]存储的某位置信息以左下角为 (0,0) 坐标，向右上角递增。Grid 类可以用于以二维数组的形式传递墙壁位置信息、食物位置信息等。其中的__str__方法可以以字符串形式输出 Pacman 地图信息，asList 方法可以输出所有值为真的位置坐标列表。

```
class Grid:
    """
    A 2-dimensional array of objects backed by a list of lists.  Data is
accessed via grid[x][y] where (x,y) are positions on a Pacman map with x
horizontal,
    """
```

1.2.4. search.py

该文件包含了所需的所有搜索方法（包括 DFS、BFS、UCS、A*算法），是 Project1 主要需要完成的部分。文件头部的 searchProblem 是一个抽象类，提供了搜索问题的框架，但未给出各个方法的具体实现（不同的 searchProblem 在 searchAgent.py 中分别定义），以供各个搜索方法使用。下方的 depthFirstSearch、breadthFirstSearch 等算法具体实现部分需要自行完成。

```
class SearchProblem:
    """
    This class outlines the structure of a search problem, but doesn't
implement
    any of the methods (in object-oriented terminology: an abstract class).
    """
```

1.2.5. searchAgent.py

该文件主要分为两部分，一部分是对 game.py 中 Agent 抽象类的继承，用以定义不同方式的智能体（如问题 8 中的贪婪找寻最近食物的智能体），这一部分也包含了 A*算法的各种启发式函数；另一部分是对 search.py 中 searchProblem 类的继承，定义了不同的搜索问题。如，问题 5、问题 6 中的问题为以最短的路径遍历迷宫的四个角，这就需要定义智能体的状态（该部分需要自行完成）；问题 7 中要以最短的路径遍历所有图中的食物，这也需要对 searchProblem 的不同定义。

searchAgent.py 文件中的各个 Problem 类定义中，有如下方法比较重要：__init__，这是对问题开始状态的初始化，可以存储墙壁位置、Pacman 的起始位置、迷宫角落等；getStartState，用于返回起始的智能体状态；isGoalState，用于判断 Pacman 是否到达目标状态；getSuccessors，用于获取当前状态的后继状态。在这四个重要的方法的实现上，对 Pacman 智能体的状态定义都尤为重要。

```
def __init__(self, startingGameState: pacman.GameState):
```

```
    """
    Stores the walls, pacman's starting position and corners.
    """
```

```
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
```

```
def isGoalState(self, state: Any):
    """
    Returns whether this search state is a goal state of the problem.
    """
```

```
def getSuccessors(self, state: Any):
    """
    Returns successor states, the actions they require, and a cost of 1.
    """
```

1.3. 解决问题的思路和想法

1.3.1. 问题 1 到问题 4 的解决思路

问题 1 到问题 4 中，需要分别实现深度优先搜索、广度优先搜索、一致代价搜索、A*搜索四种搜索方法。尽管这四种搜索算法思路不尽相同，但实现方式类似：首先将起始状态添加到对应的数据结构中，然后都是通过将当前结点的相邻结点添加到待拓展的该数据结构中，循环步骤直到找寻到正确的目标。四种算法分别使用 util.py 文件中所提供的栈、队列与优先队列实现。

1.3.2. 问题 5、问题 6 的解决思路

问题 5 的难点在于对状态的定义，若能够清楚的定义智能体在寻找角落路径问题中的状态，那么获取开始状态、获得后继状态、判断是否达到目标状态等方法的实现都较为容易了。容易思考到，Pacman 在原地图中的位置信息设计到是否有食物、是否有墙壁等问题的判断是不能省略的信息；除此之外，访问过迷宫角落中的哪些对于判断是否到达目标状态也十分重要。因此，可以定义状态为智能体在迷宫中位置与访问过的角落的坐标集合二者组成的元组。对于问题 6，找到合适的启发式函数的底线是保障其一致性，这就要求启发式函数的值要小于实际的路径代价。在本问题中，我将启发式函数定义为从当前位置到

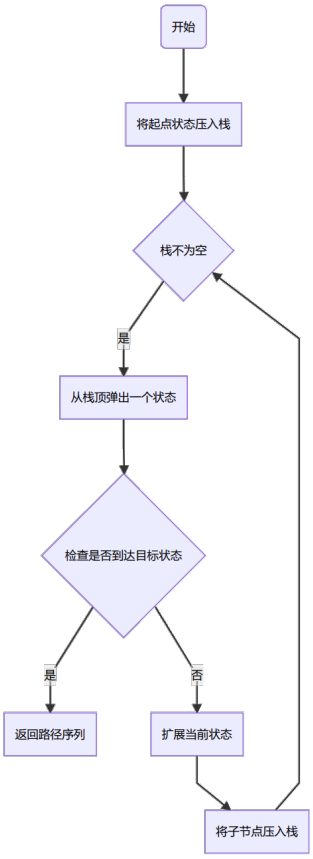
最近的角落，在依次前往智能体当下位置最近的角落直到遍历完毕的累计曼哈顿距离。

1.3.3. 问题 7、问题 8 的解决思路

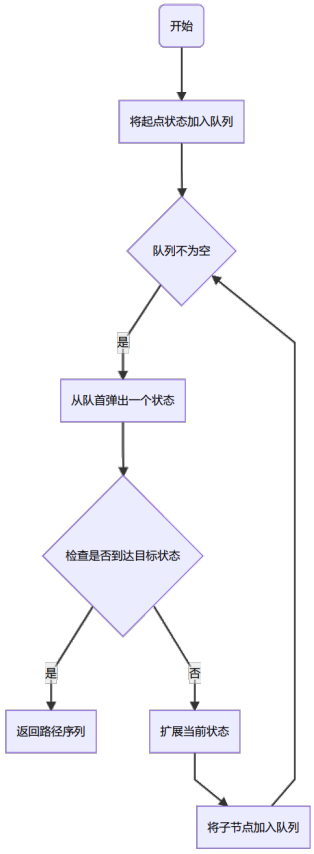
问题 7 的难点与前文中的问题 6 中相同，也在于寻找满足一致性的 A*算法启发式函数。但是由于食物的分布与角落有所不同，依次寻找曼哈顿距离最小的点并累计曼哈顿距离并不能满足一致性（实际上问题 8 中的贪婪方式也能够类似地反映出问题），因此考虑将最远的食物的实际距离作为启发式函数的值（曼哈顿距离不能很好地反应两点之间障碍的影响，使得启发式函数效果不甚理想）。对于问题 8，题目表述的较为清晰，只需要每次使用 BFS 来找寻目标，并将 isGoalState 方法设置为找到食物返回真即可。

2. 算法设计

2.1. 问题 1 深度优先搜索的实现



深度优先搜索流程图



广度优先搜索流程图

算法功能：

本算法实现了深度优先搜索算法，根据 Pacman 的起始状态（getStartState 方法获得）、目标状态（isGoalState 方法判断）以及状态转移函数（getSuccessors 方法获得）进行搜索。具体实现中，使用了一个栈（util.py 中的 Stack 类）来存储待访问的结点，将起始状态压入栈中，然后在循环中进行访问，直到栈为空为止。

设计思路：

深度优先搜索算法的设计思路是基于“回溯”思想的：在搜索过程中，从起始状态出发，一直沿着某个方向进行探索，直到不能继续前进，然后返回上一个结点，再按照其他方向继续探索，直到找到终止状态或者所有路径都已经被探索完毕。

在具体实现中，使用一个栈来存储待访问的结点，由于栈是 LIFO（后进先出）的数据结构，所以深度优先搜索算法也被称为“后序遍历”算法。每次访问一个结点时，先将其标记为已访问，然后获取其所有后继结点，并将后继结点依次压入栈中，等待下一次访问。

由于深度优先搜索算法可能会进入死循环，所以需要使用一个集合来记录已经访问过的结点，以避免重复访问同一个结点。此外，本算法实现中还需要使用一个列表来记录从起始状态到当前结点的动作序列，以便在找到终止状态时返回一系列的动作序列。

算法流程图：如上方左图所示。

2.2. 问题 2 广度优先搜索的实现

算法功能：

本算法实现了广度优先搜索算法，根据 Pacman 的起始状态（getStartState 方法获得）、目标状态（isGoalState 方法判断）以及状态转移函数（getSuccessors 方法获得）进行搜索。具体实现中，使用了一个队列（util.py 中的 Queue 类）来存储待访问的结点，将起始状态入队，然后在循环中进行访问，直到队列为空为止。

设计思路：

广度优先搜索算法的设计思路是基于“逐层扩展”思想的：在搜索过程中，从起始状态出发，依次探索所有与当前状态相邻的结点，直到找到终止状态为止。在具体实现中，使用一个队列来存储待访问的结点，由于队列是 FIFO（先进先出）的数据结构，所以广度优先搜索算法也被称为“层次遍历”算法。每次访问一个结点时，先将其标记为已访问，然后获取其所有后继结点，并将后继结点依次入队，等待下一次访问。

由于广度优先搜索算法探索的路径是从起始状态到终止状态的最短路径，所以在找到终止状态后，一定是最优解。此外，广度优先搜索算法的时间复杂度较高，但是其具有完

备性和最优性，适用于解决图的最短路径问题。

在具体实现中，本算法实现中使用列表来记录从起始状态到当前结点的动作序列，以便在找到终止状态时返回一系列的动作序列。由于广度优先搜索算法可能会探索大量的状态，所以需要使用一个集合来记录已经访问过的结点，以避免重复访问同一个结点。

算法流程图：如上方右图所示。

2.3. 问题 3 一致代价搜索的实现

算法功能：

本算法实现了 UCS（Uniform Cost Search，一致代价搜索）算法，用于在 Pacman 游戏中寻找最短路径。该算法根据 Pacman 的起始状态（getStartState 方法获得）、目标状态（isGoalState 方法判断）以及状态转移函数（getSuccessors 方法获得）进行搜索，并使用优先队列（util.py 中的 PriorityQueue 类）来存储待访问的结点，其中结点的优先级由累计代价和确定。在算法执行过程中，每次从队列中取出代价最小的结点进行访问，直到找到终止状态或者队列为空为止。

设计思路：

UCS 算法是一种基于 Dijkstra 算法的启发式搜索算法，其基本思路是在每次扩展结点时，优先考虑代价最小的路径。该算法使用一个优先队列来存储待访问的结点，其中结点的优先级由累计代价和确定。在具体实现中，每次从队列中取出代价最小的结点进行访问，然后获取其所有后继结点，并将后继结点依次加入队列中，等待下一次访问。如果队列为空，则表示无法到达目标状态。

由于 UCS 算法使用了优先队列，每次取出代价最小的结点进行访问，所以它可以保证找到的路径是代价最小的路径。与深度优先搜索、广度优先搜索类似，也需要用一个集合来记录访问过的状态避免陷入死循环中。此外在具体实现中，同样需要使用列表来记录从起始状态到当前结点的动作序列，以便在找到终止状态时返回一系列的动作序列。在找到终止状态时，可以根据该列表得到从起始状态到终止状态的最短路径（动作序列）。

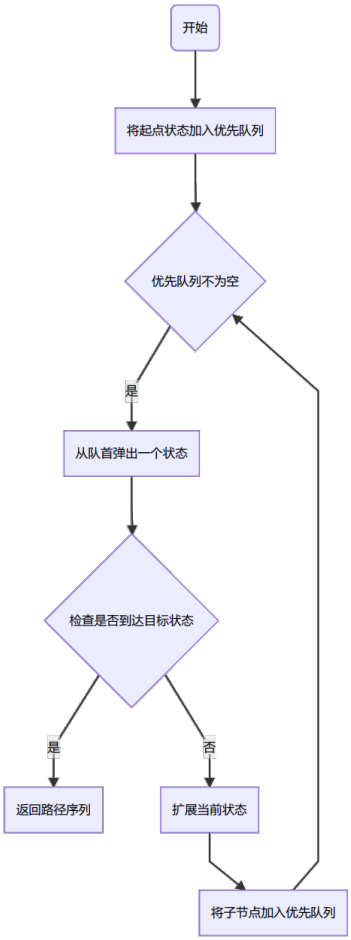
算法流程图：如下图所示（与 A*算法相同，实际上即 h 为 0 的 A*算法）。

2.4. 问题 4 A*搜索的实现

算法功能：

本算法实现了 A*搜索算法，根据 Pacman 的起始状态（getStartState 方法获得）、目标状态（isGoalState 方法判断）以及状态转移函数（getSuccessors 方法获得）进行搜索。在

搜索过程中，采用了一个启发式函数（本问题中使用曼哈顿距离）来指导搜索方向，以尽可能快地找到最优解。具体实现中，使用了一个优先队列（util.py 中的 PriorityQueue 类）来存储待访问的结点，将起始状态压入队列中，然后在循环中进行访问，直到队列为空或者找到终止状态为止。



一致代价搜索与 A*搜索的算法流程

设计思路：

A 搜索算法是一种综合利用启发式函数和广度优先搜索的算法，它尽可能快地找到最优解，同时保证时间复杂度尽可能小。具体实现中，使用一个优先队列来存储待访问的结点，每次从队列中取出距离起始状态最近的结点进行扩展，并将扩展后的结点依次加入优先队列中。在加入队列时，需要计算每个结点的 f 值（f 值表示从起始状态到该结点的实际代价（g 值）和该结点到目标状态的估计代价（h 值）之和）。本问题中的启发式函数采用曼哈顿距离，即将当前结点到目标状态的每一维坐标之差取绝对值后求和。若当前状态坐标为 (x_1, y_1)，目标状态坐标为 (x_2, y_2)，则曼哈顿距离定义的启发式函数为：

$$h = |x_1 - x_2| + |y_1 - y_2|$$

同前面的算法类似，所以需要使用一个集合来记录已经访问过的结点，以避免重复访问同一个结点而陷入死循环。此外，本算法实现中还需要使用一个列表来记录从起始状态到当前结点的动作序列，以便在找到终止状态时返回一系列的动作序列。

算法流程图：如上图所示（与 UCS 算法相同）。

2.5. 问题 5 迷宫 4 角落的 BFS 寻路

算法功能：

该算法实现了在 Pacman 游戏中，从起始位置开始遍历迷宫四个角的最短路径。为了实现这一目标，算法定义了一个状态空间，并运用了前文所述的问题 2 中的深度优先搜索算法在此状态空间上实现了寻找最短路径的搜索算法。

设计思路：

1. 状态空间的定义：

在该算法中，定义的状态空间包括两个部分：智能体在迷宫中的位置信息和访问过的角落的坐标集合。为了方便实现和计算，这两部分信息被组成了一个元组作为状态的表示方式。在状态空间的定义中，通过元组中的位置信息来表示智能体在迷宫中的位置信息，通过元组中的访问过的角落的坐标集合来表示智能体已经访问过的角落。

2. 后继状态的计算：

在状态空间中，一个状态的后继状态即是智能体从当前位置所能够移动到的位置，以及访问角落的状态是否发生改变。为了计算后继状态，算法首先获取当前状态中智能体的位置信息，然后通过枚举所有可能的移动方向，计算智能体移动后的新位置。在计算新位置时，如果新位置没有被障碍物所占据，则将该新位置和访问角落的状态加入到后继状态的集合中。在后继状态的计算中，如果新位置为迷宫中的一个角落，算法还需要将该角落的坐标加入到访问角落的状态中。这样，在搜索算法中，当访问到的角落的坐标集合包含了所有四个角的坐标时，就说明已经找到了从起始位置开始遍历迷宫四个角的最短路径。

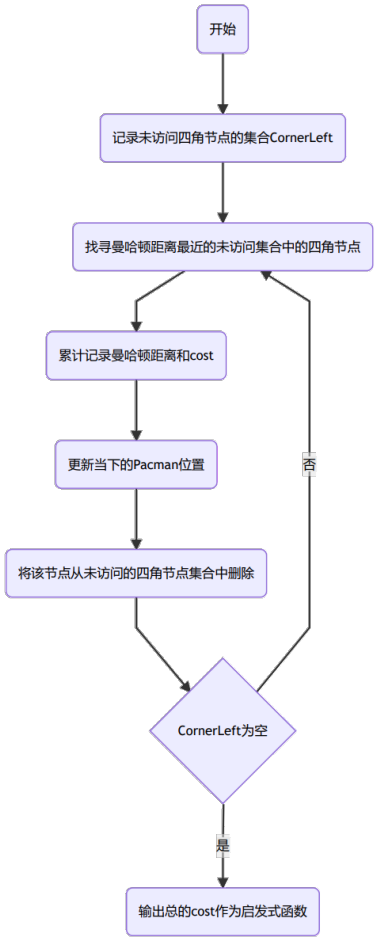
3. 目标状态的判断：

在该算法中，目标状态即是访问过所有四个角的状态。在目标状态的判断中，算法只需要判断当前状态中访问过的角落的数量是否等于四即可。如果等于四，说明已经找到了从起始位置开始遍历迷宫四个角的最短路径，当前状态即为目标状态。

2.6. 问题 6 迷宫 4 角落的 A*算法启发式函数定义

算法功能：

该算法应对的问题与问题 5 中相同，均是寻找从起始位置开始遍历迷宫四个角的最短路径。本题中使用前文中提到的 A*算法进行寻路，关键点在于找到合适的启发式函数。



问题 6 中 A*算法的启发式函数算法流程图

设计思路：

1. 首先，获取迷宫地图中四个角的坐标。
2. 与问题 5 相同，定义搜索状态表示为(state, cornersVisited)，其中 state 表示当前 Agent 在地图中的位置，集合 cornersVisited 表示 Agent 已经访问过的四个角。
3. 定义一个集合 cornersLeft，其中包含所有还没有被访问的四个角。
4. 对 cornersLeft 中的每个角进行评估，评估的依据是距离当前 Agent 位置最近的角，这里采用曼哈顿距离（Manhattan distance）来计算距离。
5. 选择曼哈顿距离值最高的角，累计计算总代价 cost，将 Agent 移动到所选的角位置（不做实际移动，只是方便计算的近似理解），并将该角从 cornersLeft 中删除。
6. 重复步骤 4 和 5，直到 cornersLeft 为空集。

7. 算法的返回值是曼哈顿距离的累加和，即从起始位置到访问完所有角的最短路径长度。

容易得到，该方式求得的启发式函数值是当迷宫里不存在任何阻碍时 Pacman 智能体的最短路径，因此在有障碍的迷宫情况下能够满足启发式函数一致性的条件。在完成了之后问题的小题后，我受启发尝试了以最远的角落中到起始状态位置的曼哈顿距离作为启发式函数，也能够效果不错地解决问题，这在报告之后的实现部分将会详细介绍。

算法流程图：见上方图。

2.7. 问题 7 多个食物 A*算法启发式函数定义

算法功能：

本题的算法需要为 A*算法定义合适的启发式函数来进行路径规划，从而让 Pacman 智能体实现以最短的路径遍历完迷宫中所有的食物。

设计思路：

若用问题 6 中的方式来定义启发式函数，在多个食物分布在地图多处的情况下类似于问题 8 中贪婪地选择最近的食物并不是好的策略，且曼哈顿距离在迷宫地形比较复杂的情况下不能很好的反应点到点的实际距离；累计的 cost 值可能会超出实际从当前位置到目标位置的代价，导致启发式函数不能满足要求，让 A*算法无法实现最优的路径规划。

考虑到曼哈顿距离不能很好地反映两点之间在障碍阻隔下的实际距离，可以尝试使用 BFS 搜索到的实际距离来代表两点之间的距离。尝试了选择最近食物的距离以及最远食物的距离后，发现后者作为启发式函数的效果更好。直观上，以最远食物的距离作为启发式函数也较为合理，因为它大概代表了最后 Pacman 完成目标所花费的代价。

2.8. 问题 8 贪婪式选取最近的食物

算法功能：

本题实现一个贪婪搜索算法，将搜索目标定为贪婪地找到距离当前搜索点位最近的食物点位。具体来说，算法会从当前搜索点出发，找到距离当前点最近的一个食物点，然后将该点作为下一个搜索点，继续进行搜索，直到所有食物点都被找到为止。

设计思路：

本题的算法实现较为简单。在寻路问题上，只需调用问题 2 中已经实现的 BFS 搜索算法，将目标状态定义为任意的食物位置即可。广度优先搜索搜索到的第一个食物位置即为贪婪搜索所要求的最近的食物。

3. 算法实现

3.1. 问题 1 深度优先搜索的实现

在数据结构使用方面，使用 util.py 中实现的 Stack 栈（LIFO）来存储待访问的结点，并用一个集合 visit 来记录已经访问过的结点，从而防止重复访问造成的死循环。

算法开始时，先将初始状态压入栈中以开始搜索。通过 s.push 方法将一个三项元组压入栈中，其中三项分别对应当前坐标（初始坐标）、遍历的动作序列（一个列表，初始为空）、遍历的总代价和（初始为 0）。

若栈不为空，则循环进行下列操作：从栈顶弹出一个结点信息，若其通过 isGoalState 方法确认为目标位置，则返回一系列的动作序列（即栈中元素的第二项列表）。每次访问一个结点时，若其已经访问过（在 visit 中）则不进行操作；否则，先通过将其坐标加入 visit 集合中标记为已访问，然后获取其所有后继结点，并将后继结点依次压入栈中，等待下一次访问。压入的后继结点信息中，第一项坐标是 getSuccessors 方法所获得的，第二项的动作序列要在原当前结点的列表末尾增加新的一步动作，第三项的累计代价和为当前结点的累计代价和加上转移代价（本问题为 1）。

```
def depthFirstSearch(problem: SearchProblem):
    "*** YOUR CODE HERE ***"
    from util import Stack
    s = Stack()         # 深度优先搜索使用栈来记录待访问的结点（LIFO）
    visit = set()       # 使用一个集合 visit 来记录已经访问过的坐标位置
    # 三项元组分别对应(坐标, 遍历的动作序列, 累计代价和)
    s.push((problem.getStartState(), [], 0))

    while(s.isEmpty() == 0):
        cur = s.pop()

        # 若到达了目标位置，则返回一系列的动作序列
        if(problem.isGoalState(cur[0])):
            return cur[1]

        # 已经访问过的位置就跳过
        if(cur[0] not in visit):
            # 标记访问
            visit.add(cur[0])
            # 获取下一个结点
```

```
            success = problem.getSuccessors(cur[0])
            for suc in success:
                # 三项元组分别对应(后继位置的坐标, 遍历的动作序列, 累计代价和)
                s.push((suc[0], cur[1] + [suc[1]], cur[2] + suc[2]))
```

3.2. 问题 2 广度优先搜索的实现

在数据结构使用方面，使用 util.py 中实现的 Queue 队列（FIFO）来存储待访问的结点，并用一个集合 visit 来记录已经访问过的结点，从而防止重复访问造成的死循环。

算法开始时，先将初始状态加入队列中以开始搜索。通过 q.push 方法将一个三项元组加入队列中，其中三项分别对应当前坐标（初始坐标）、遍历的动作序列（一个列表，初始为空）、遍历的总代价和（初始为 0）。

若队列不为空，则循环进行下列操作：从队首弹出一个结点信息，若其通过 isGoalState 方法确认为目标位置，则返回一系列的动作序列（即队列中元素的第二项列表）。每次访问一个结点时，若其已经访问过（在 visit 中）则不进行操作；否则，先通过将其坐标加入 visit 集合中标记为已访问，然后获取其所有后继结点，并将后继结点依次加入队列中，等待下一次访问。加入的后继结点信息中，第一项坐标是 getSuccessors 方法所获得的，第二项的动作序列要在原当前结点的列表末尾增加新的一步动作，第三项的累计代价和为当前结点的累计代价和加上转移代价（本问题为 1）。

```
def breadthFirstSearch(problem: SearchProblem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    from util import Queue
    q = Queue()         # 广度优先搜索使用队列来记录待访问的结点（FIFO）
    visit = set()       # 使用一个集合 visit 来记录已经访问过的坐标位置
    # 三项元组分别对应(坐标, 遍历的动作序列, 累计代价和)
    q.push((problem.getStartState(), [], 0))

    while(q.isEmpty() == 0):
        cur = q.pop()

        # 若到达了目标位置，则返回一系列的动作序列
        if(problem.isGoalState(cur[0])):
            return cur[1]

        # 已经访问过的位置就跳过
```

```
        if(cur[0] not in visit):
            # 标记访问
            visit.add(cur[0])
            # 获取下一个结点
            success = problem.getSuccessors(cur[0])
            for suc in success:
                # 三项元组分别对应(后继位置的坐标, 遍历的动作序列, 累计代价和)
                q.push((suc[0], cur[1] + [suc[1]], cur[2] + suc[2]))
```

3.3. 问题 3 一致代价搜索的实现

在数据结构使用方面，使用 util.py 中实现的 PriorityQueue 队列（队首元素为值最小的元素）来存储待访问的结点，并用一个集合 visit 来记录已经访问过的结点，从而防止重复访问造成的死循环。

算法开始时，先将初始状态加入队列中以开始搜索。通过 q.push 方法将一个二项元组（其中第一项为类似问题 1、问题 2 中的三项元组，三项分别对应当前坐标（初始坐标）、遍历的动作序列（一个列表，初始为空）、遍历的总代价和（初始为 0）；后一项为 PriorityQueue 所需要的排序依据，即累计代价和）加入队列中。

若队列不为空，则循环进行下列操作：从队首弹出一个结点信息，若其通过 isGoalState 方法确认为目标位置，则返回一系列的动作序列（即队列中元素的第二项列表）。每次访问一个结点时，若其已经访问过（在 visit 中）则不进行操作；否则，先通过将其坐标加入 visit 集合中标记为已访问，然后获取其所有后继结点，并将后继结点依次加入队列中，等待下一次访问。加入的后继结点信息中，三元组的第一项坐标是 getSuccessors 方法所获得的，第二项的动作序列要在原当前结点的列表末尾增加新的一步动作，第三项的累计代价和为当前结点的累计代价和加上转移代价（本问题为 1）。

```
def uniformCostSearch(problem: SearchProblem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    from util import PriorityQueue
    q = PriorityQueue()     # 一致代价搜索使用优先队列来记录待访问的结点（优先级为路径的累计代价）
    visit = set()           # 使用一个集合 visit 来记录已经访问过的坐标位置
    # 三项元组分别对应(坐标, 遍历的动作序列, 累计代价和); 后面的 0 为 PriorityQueue 排序的依据，即代价和
    q.push((problem.getStartState(), [], 0), 0)
```

```
    while(q.isEmpty() == 0):
        cur = q.pop()

        # 若到达了目标位置，则返回一系列的动作序列
        if(problem.isGoalState(cur[0])):
            return cur[1]

        # 已经访问过的位置就跳过
        if(cur[0] not in visit):
            # 标记访问
            visit.add(cur[0])
            # 获取下一个结点
            success = problem.getSuccessors(cur[0])
            for suc in success:
                # 三项元组分别对应(后继位置的坐标，遍历的动作序列，累计代价和)；后面的一项为 PriorityQueue 排序的依据，即代价和
                q.push((suc[0], cur[1] + [suc[1]], cur[2] + suc[2]), cur[2] + suc[2])
```

3.4. 问题 4 A*搜索的实现

在数据结构使用方面，使用 util.py 中实现的 PriorityQueue 队列（队首元素为值最小的元素）来存储待访问的结点，并用一个集合 visit 来记录已经访问过的结点，从而防止重复访问造成的死循环。

算法开始时，先将初始状态加入队列中以开始搜索。通过 q.push 方法将一个二项元组（其中第一项为类似问题 1、问题 2 中的三项元组，三项分别对应当前坐标（初始坐标）、遍历的动作序列（一个列表，初始为空）、遍历的总代价和（初始为 0）；后一项为 PriorityQueue 所需要的排序依据，即累计代价与启发式函数的和）加入队列中。

若队列不为空，则循环进行下列操作：从队首弹出一个结点信息，若其通过 isGoalState 方法确认为目标位置，则返回一系列的动作序列（即队列中元素的第二项列表）。每次访问一个结点时，若其已经访问过（在 visit 中）则不进行操作；否则，先通过将其坐标加入 visit 集合中标记为已访问，然后获取其所有后继结点，并将后继结点依次加入队列中，等待下一次访问。加入的后继结点信息中，三元组的第一项坐标是 getSuccessors 方法所获得的，第二项的动作序列要在原当前结点的列表末尾增加新的一步动作，第三项的累计代价和为当前结点的累计代价和加上转移代价（本问题为 1）。同样，还需要在之后加上一项作为 PriorityQueue 所需要的排序依据，即累计代价与启发式函数的和（启发式函数通过 heuristic 变量传入）。

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic
first."""
    "*** YOUR CODE HERE ***"
    from util import PriorityQueue
    q = PriorityQueue()          # 一致代价搜索使用优先队列来记录待访问的结点（优先
级为路径的累计代价）
    visit = set()                # 使用一个集合 visit 来记录已经访问过的坐标位置
    # 三项元组分别对应(坐标, 遍历的动作序列, 累计代价和); 后面的 0 为 PriorityQueue
排序的依据，即 f = g + h
    q.push((problem.getStartState(), [], 0), 0)

    while(q.isEmpty() == 0):
        cur = q.pop()

        # 若到达了目标位置，则返回一系列的动作序列
        if(problem.isGoalState(cur[0])):
            return cur[1]

        # 已经访问过的位置就跳过
        if(cur[0] not in visit):
            visit.add(cur[0])
            success = problem.getSuccessors(cur[0])
            for suc in success:
                # 三项元组分别对应(后继位置的坐标, 遍历的动作序列, 累计代价和); 后
面的一项为 PriorityQueue 排序的依据，，即 f = g + h
                q.push((suc[0], cur[1] + [suc[1]], cur[2] + suc[2]), cur[2] +
suc[2] + heuristic(suc[0], problem))
```

3.5. 问题 5 迷宫 4 角落的 BFS 寻路

如前文所述，该问题中定义的状态空间包括两个部分：智能体在迷宫中的位置信息和访问过的角落的坐标集合。为了方便实现和计算，这两部分信息被组成了元组作为状态的表示方式。在状态空间的定义中，通过元组中的位置信息来表示智能体在迷宫中的位置信息，通过元组中的访问过的角落的坐标集合来表示智能体已经访问过的角落。

根据状态的定义，完善 getStartState、isGoalState 以及 getSuccessors 三个方法。在 getStartState 中，返回一个如上述的二项元组，其中第一项是 Pacman 的起始地图位置，后一项是一个空的 frozenset 集合；在 isGoalState 中，通过已经访问过角落集合的长度来判断是否到达目标状态；getSuccesors 中仿照问题 1-4 中 Agent 的书写方式，只是在后继状态的

产生上要对应的改变记录角落的访问状态的集合，因此对于每个新拓展的后继结点判断其是否是迷宫角落，若是则在访问过的角落集合中加入该结点。

```
    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman
state
        space)
        """
        "*** YOUR CODE HERE ***"
        # 开始状态包含起始位置，以及访问过的集合（空）
        return self.startingPosition, frozenset()
```

```
    def isGoalState(self, state: Any):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        # 若四个结点都访问过，则达到目标状态
        return len(state[1]) == 4
```

```
    def getSuccessors(self, state: Any):
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:
            # Add a successor state to the successor list if the action is
legal
            # Here's a code snippet for figuring out whether a new position
hits a wall:
            x,y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            "*** YOUR CODE HERE ***"
            # 仿照问题 1-4 的 Agent 书写
            if not self.walls[nextx][nexty]:
                nextPos = (nextx, nexty)
                # 如果走到了新的角落就更新访问集合
                if(nextPos in self.corners):
                    corners = state[1] | {nextPos}
                # 如果没有走到新的角落就保持集合不变
                else:
```

```
                corners = state[1]
            # 三元组是(后继状态, 转移动作, 代价)
            successors.append( ( (nextPos, corners), action, 1) )
        self._expanded += 1 # DO NOT CHANGE
        return successors
```

3.6. 问题 6 迷宫 4 角落的 A*算法启发式函数定义

如前文介绍，问题 5 中定义的状态（即此处传入的状态）是一个二项元组，用 position 与 cornersVisited 来接受 Pacman 智能体在地图中的位置与访问过的角落信息。cornersLeft 集合记录未访问过的角落坐标集合，即以 corners 与 cornersVisited 做差进行初始化。当 cornersLeft 不为空时，循环进行如下操作：使用 min 函数求出曼哈顿距离最近的角落坐标，累计计算代价值；更新 position 为选出的曼哈顿距离最近角落，并将其从 cornersLeft 集合中移除。最终得到的 cost 即为启发式函数的值。

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    corners = problem.corners # These are the corner coordinates

    "*** YOUR CODE HERE ***"
    position        = state[0]       # 当前 Agent 在地图中的位置
    cornersVisited  = state[1]       # Agent 访问过的角落
    cost = 0                        # 计算启发式函数的代价

    cornersLeft = set(corners) - cornersVisited     # Agent 还没访问过的角落

    # 计算的思路是依次选择曼哈顿距离最近的角落访问并累计曼哈顿距离，更新 Agent 位置寻找下一个角落直到访问完毕
    while(len(cornersLeft) != 0):
        minCorner = min(cornersLeft, key = lambda x : abs(position[0] - x[0]) + abs(position[1] - x[1]))    # 找到曼哈顿距离最近的角落
        cost = cost + abs(position[0] - minCorner[0]) + abs(position[1] - minCorner[1])     # 更新花费值
        cornersLeft.remove(minCorner)   # 去除访问后的结点
        position = minCorner            # 更新当前位置

    return cost # Default to trivial solution
```

在完成了问题 7 中的启发式函数后，我也受到启发想到了思路类似且更为便捷的启发式函数，即选取曼哈顿距离最远的角落坐标作为启发式函数的值。在找寻曼哈顿距离最远

的角落时方法与前者类似，具体的代码如下：

```
def cornersHeuristic(state: Any, problem: CornersProblem):
    corners = problem.corners # These are the corner coordinates
    "*** YOUR CODE HERE ***"
    cost = 0                          # 计算启发式函数的代价
    position       = state[0]        # 当前 Agent 在地图中的位置
    cornersVisited = state[1]        # Agent 访问过的角落
    cornersLeft = set(corners) - cornersVisited     # Agent 还没访问过的角落
    # 所有角落都访问过了
    if(len(cornersLeft) == 0):
        return 0

    maxCorner = max(cornersLeft, key = lambda x : abs(position[0] - x[0]) +
abs(position[1] - x[1]))    # 找到曼哈顿距离最远的角落
    cost = cost + abs(position[0] - maxCorner[0]) + abs(position[1] -
maxCorner[1])               # 将曼哈顿距离作为答案

    return cost # Default to trivial solution
```

3.7. 问题 7 多个食物 A*算法启发式函数定义

如算法设计部分所述，选择将最远的食物到 Pacman 的距离作为启发式函数的值。实际上所有的食物距离通过对全地图的一次 BFS 的搜索就能全部得到，且最后搜索到的食物距离就是最远的食物到 Pacman 智能体的距离。因此可以仿照问题 2 与 getSuccessors 方法书写如下代码，将 cost 不断更新为最新的食物距离（BFS 的搜索深度）即可。

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem:
FoodSearchProblem):
    position, foodGrid = state
    foodPos = foodGrid.asList()             # 地图上所有食物的位置
    cost = 0                                # 启发式函数的值

    from util import Queue
    q = Queue()
    visit = set()
    q.push((position, 0))

    while(q.isEmpty() == 0):
        cur = q.pop()
```

```
        # 没有搜索过的结点再进行
        if(cur[0] not in visit):
            visit.add(cur[0])
            # 最后搜到的食物是最远的 cost 最大
            if(cur[0] in foodPos):
                cost = cur[1]

            # 这部分求算后继结点
            success = []
            for dx, dy in [(1, 0), (-1, 0), (0, -1), (0, 1)]:
                newx, newy = cur[0][0] + dx, cur[0][1] + dy
                # 排除有墙壁的非法结点
                if problem.walls[newx][newy]:
                    continue
                # 排除位置不在地图内的非法结点
                elif newx < 0 or newy < 0 or newx >= problem.walls.width or
newy >= problem.walls.height:
                    continue
                # 准备拓展该结点
                else:
                    success.append((newx, newy))

            # 将合法后继结点加入队列
            for suc in success:
                q.push((suc, cur[1] + 1))
    return cost
```

在相同的思路下，如果使用 searchAgent.py 中的 mazeDistance 函数，则可以使代码更为简洁。但是由于要依次调用 BFS 搜索迷宫中到每一个食物的路径，而非一次性使用 BFS 更新最远的食物距离，这会导致搜索效率的降低。代码如下：

```
def foodHeuristic(state: Tuple[Tuple, List[List]], problem:
FoodSearchProblem):
    position, foodGrid = state
    foodPos = foodGrid.asList()              # 地图上所有食物的位置
    cost = 0                                 # 启发式函数的值

    for food in foodPos:
```

```
        dis = mazeDistance(position, food, problem.startingGameState)
        cost = cost if cost > dis else dis
    return cost
```

3.8. 问题 8 贪婪式选取最近的食物

本题的实现最为简单，只需每次用 BFS 进行搜索，并将目标状态设定为若该位置有食物即为目标状态即可。代码如下：

```
def findPathToClosestDot(self, gameState: pacman.GameState):
    """
    Returns a path (a list of actions) to the closest dot, starting from
    gameState.
    """
    # Here are some useful elements of the startState
    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)

    "*** YOUR CODE HERE ***"
    # 使用 bfs 搜索能迅速的找到最近的食物
    return search.bfs(problem)
```

```
def isGoalState(self, state: Tuple[int, int]):
    x,y = state
    "*** YOUR CODE HERE ***"
    # 若当前位置有食物那么就是目标位置
    return self.food[x][y]
```

4. 实验结果

4.1. 问题 1 深度优先搜索的实现

测试深度优先搜索在小、中、大地图中的表现情况：

tinyMaze	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l tinyMaze -p SearchAgent [SearchAgent] using function depthFirstSearch [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 10 in 0.0 seconds Search nodes expanded: 15 Pacman emerges victorious! Score: 500 Average Score: 500.0 Scores: 500.0 Win Rate: 1/1 (1.00) Record: Win
mediumMaze	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l mediumMaze -p SearchAgent [SearchAgent] using function depthFirstSearch [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 130 in 0.0 seconds Search nodes expanded: 146 Pacman emerges victorious! Score: 380 Average Score: 380.0 Scores: 380.0 Win Rate: 1/1 (1.00) Record: Win
bigMaze	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l bigMaze -z .5 -p SearchAgent [SearchAgent] using function depthFirstSearch [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 210 in 0.0 seconds Search nodes expanded: 390 Pacman emerges victorious! Score: 300 Average Score: 300.0 Scores: 300.0 Win Rate: 1/1 (1.00) Record: Win

对于所有的测试样例均通过，截图如下：

```
Question q1
===========
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'D', 'C']
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
***     solution:               ['2:A->D', '0:D->G']
***     expanded_states:        ['A', 'D']
*** PASS: test_cases\q1\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_manypaths.test
***     solution:               ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']
***     expanded_states:        ['A', 'B2', 'C', 'D', 'E2', 'F']
*** PASS: test_cases\q1\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 130
***     nodes expanded:         146

### Question q1: 3/3 ###


Finished at 11:42:19

Provisional grades
==================
Question q1: 3/3
------------------
Total: 3/3
```

4.2. 问题 2 广度优先搜索的实现

测试广度优先搜索在中、大地图中的表现情况：

mediumMaze	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs [SearchAgent] using function bfs [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 68 in 0.0 seconds Search nodes expanded: 269 Pacman emerges victorious! Score: 442 Average Score: 442.0 Scores: 442.0 Win Rate: 1/1 (1.00) Record: Win

bigMaze	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5 [SearchAgent] using function bfs [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 210 in 0.0 seconds Search nodes expanded: 620 Pacman emerges victorious! Score: 300 Average Score: 300.0 Scores: 300.0 Win Rate: 1/1 (1.00) Record: Win

对于类似于华容道的 8 puzzle 问题，BFS 也能给出正确的最小步数解：

```
PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python eightpuzzle.py
A random puzzle:
-------------
| 3 | 2 | 5 |
-------------
| 6 | 1 | 8 |
-------------
| 7 |   | 4 |
-------------
BFS found a path of 9 moves: ['right', 'up', 'up', 'left', 'down', 'down', 'left', 'up', 'up']
```

对于所有的测试样例均通过，截图如下：

```
Question q2
===========
*** PASS: test_cases\q2\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q2\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q2\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q2\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 3/3 ###


Finished at 11:50:18

Provisional grades
==================
Question q2: 3/3
------------------
Total: 3/3
```

4.3. 问题 3 一致代价搜索的实现

测试一致代价搜索在不同地形的中地图中的表现情况。由于普通地图每一步代价为 1，拓展结点数较 BFS 没有变化；值得注意的是，在迷宫 mediumDottedMaze 和 mediumScaryMaze 中，分别有多个食物以及幽灵的场景，通过给定的不同情况下的 cost 函数，均使用 UCS 在拓展结点较小的情况下找到了最佳的路径。

mediumMaze	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs [SearchAgent] using function ucs [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 68 in 0.0 seconds Search nodes expanded: 269 Pacman emerges victorious! Score: 442 Average Score: 442.0 Scores: 442.0 Win Rate: 1/1 (1.00) Record: Win

mediumDotted Maze	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l mediumDottedMaze -p StayEastSearchAgent Path found with total cost of 1 in 0.0 seconds Search nodes expanded: 186 Pacman emerges victorious! Score: 646 Average Score: 646.0 Scores: 646.0 Win Rate: 1/1 (1.00) Record: Win
mediumScary Maze	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l mediumScaryMaze -p StayWestSearchAgent Path found with total cost of 68719479864 in 0.0 seconds Search nodes expanded: 108 Pacman emerges victorious! Score: 418 Average Score: 418.0 Scores: 418.0 Win Rate: 1/1 (1.00) Record: Win

对于所有的测试样例均通过，截图如下：

```
Question q3
===========
*** PASS: test_cases\q3\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q3\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q3\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q3\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q3\ucs_0_graph.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q3\ucs_1_problemC.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269
*** PASS: test_cases\q3\ucs_2_problemE.test
***     pacman layout:          mediumMaze
***     solution length: 74
***     nodes expanded:         260
*** PASS: test_cases\q3\ucs_3_problemW.test
***     pacman layout:          mediumMaze
***     solution length: 152
***     nodes expanded:         173
*** PASS: test_cases\q3\ucs_4_testSearch.test
***     pacman layout:          testSearch
***     solution length: 7
***     nodes expanded:         14
*** PASS: test_cases\q3\ucs_5_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']

### Question q3: 3/3 ###
```

4.4. 问题 4 A*搜索的实现

测试 A*搜索在大地图中的表现情况，可以观察到在使用曼哈顿距离作为启发式函数的基础上，A*算法较 BFS 少拓展了近百个结点，提升了搜索效率：

bigMaze	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic [SearchAgent] using function astar and heuristic manhattanHeuristic [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 210 in 0.0 seconds Search nodes expanded: 549 Pacman emerges victorious! Score: 300 Average Score: 300.0 Scores: 300.0 Win Rate: 1/1 (1.00) Record: Win

对于所有的测试样例均通过，截图如下：

```
Question q4
===========
*** PASS: test_cases\q4\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q4\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases\q4\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases\q4\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q4\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q4\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###
```

4.5. 问题 5 迷宫 4 角落的 BFS 寻路

测试 BFS 在小、中四角地图中的表现情况：

tinyCorners	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem [SearchAgent] using function bfs [SearchAgent] using problem type CornersProblem Path found with total cost of 28 in 0.0 seconds Search nodes expanded: 252 Pacman emerges victorious! Score: 512 Average Score: 512.0 Scores: 512.0 Win Rate: 1/1 (1.00) Record: Win
mediumCorners	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem [SearchAgent] using function bfs [SearchAgent] using problem type CornersProblem Path found with total cost of 106 in 0.0 seconds Search nodes expanded: 1966 Pacman emerges victorious! Score: 434 Average Score: 434.0 Scores: 434.0 Win Rate: 1/1 (1.00) Record: Win

对于所有的测试样例均通过，截图如下(Question q2 部分前文给出，此处省略)：

```
Question q5
===========
*** PASS: test_cases\q5\corner_tiny_corner.test
***     pacman layout:          tinyCorner
***     solution length:                28

### Question q5: 3/3 ###


Finished at 12:21:13

Provisional grades
==================
Question q2: 3/3
Question q5: 3/3
------------------
Total: 6/6
```

4.6. 问题 6 迷宫 4 角落的 A*算法启发式函数定义

分别测试前文中提到的两种启发式函数在大地图中的表现，可以观察到前者拓展的结

点数显著更少，作为启发式函数的效果更佳。

最近累计 曼哈顿距离	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5 Path found with total cost of 106 in 0.0 seconds Search nodes expanded: 692 Pacman emerges victorious! Score: 434 Average Score: 434.0 Scores: 434.0 Win Rate: 1/1 (1.00) Record: Win
最远 曼哈顿距离	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5 Path found with total cost of 106 in 0.0 seconds Search nodes expanded: 1136 Pacman emerges victorious! Score: 434 Average Score: 434.0 Scores: 434.0 Win Rate: 1/1 (1.00) Record: Win

对于所有的测试样例均通过，截图如下(Question q4 部分前文给出，此处省略)：

```
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West', 'West', 'West', 'West', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'West
', 'West', 'South', 'South', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'South', 'South', 'West', 'West', 'South', 'South', 'South', 'West', 'West', 'East', 'East', '
North', 'North', 'North', 'East', 'East', 'East', 'East', 'East', 'East', 'East', 'East', 'South', 'South', 'East', 'East', 'East', 'East', 'East', 'North', 'North', 'East', 'East', 'North',
'North', 'East', 'East', 'North', 'North', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'East', 'East', 'North', 'North', 'East', 'East', 'South', 'South', 'South', 'So
uth', 'South', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'North', 'North', 'East', 'East', 'North', 'North']
path length: 106
*** PASS: Heuristic resulted in expansion of 692 nodes

### Question q6: 3/3 ###


Finished at 12:30:41

Provisional grades
==================
Question q4: 3/3
Question q6: 3/3
------------------
Total: 6/6
```

4.7. 问题 7 多个食物 A*算法启发式函数定义

在普通小迷宫和棘手的迷宫中测试启发式函数。注意到，在 trickySearch 中若使用 UCS（即启发式函数为 0），要拓展超过 16000 个结点；在使用了如前文所述的 A*算法后，拓展结点数大幅减少到 4137，启发式函数能够很好的起到作用。

testSearch	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l testSearch -p AStarFoodSearchAgent Path found with total cost of 7 in 0.0 seconds Search nodes expanded: 10 Pacman emerges victorious! Score: 513 Average Score: 513.0 Scores: 513.0 Win Rate: 1/1 (1.00) Record: Win
trickySearch	PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l trickySearch -p AStarFoodSearchAgent Path found with total cost of 60 in 2.0 seconds Search nodes expanded: 4137 Pacman emerges victorious! Score: 570 Average Score: 570.0 Scores: 570.0 Win Rate: 1/1 (1.00) Record: Win

对于所有的测试样例均通过，截图如下(Question q4 部分前文给出，此处省略)：

```
Question q7
===========
*** PASS: test_cases\q7\food_heuristic_1.test
*** PASS: test_cases\q7\food_heuristic_10.test
*** PASS: test_cases\q7\food_heuristic_11.test
*** PASS: test_cases\q7\food_heuristic_12.test
*** PASS: test_cases\q7\food_heuristic_13.test
*** PASS: test_cases\q7\food_heuristic_14.test
*** PASS: test_cases\q7\food_heuristic_15.test
*** PASS: test_cases\q7\food_heuristic_16.test
*** PASS: test_cases\q7\food_heuristic_17.test
*** PASS: test_cases\q7\food_heuristic_2.test
*** PASS: test_cases\q7\food_heuristic_3.test
*** PASS: test_cases\q7\food_heuristic_4.test
*** PASS: test_cases\q7\food_heuristic_5.test
*** PASS: test_cases\q7\food_heuristic_6.test
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** PASS: test_cases\q7\food_heuristic_grade_tricky.test
***     expanded nodes: 4137
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 5/4 ###
```

4.8. 问题 8 贪婪式选取最近的食物

先测试贪婪式搜索在全是食物的食物的迷宫中的情况：

```
PS C:\Users\BoyuanZheng\Desktop\人工智能\search> python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with cost 350.
Pacman emerges victorious! Score: 2360
Average Score: 2360.0
Scores:        2360.0
Win Rate:      1/1 (1.00)
Record:        Win
```

对于所有的测试样例均能通过：

```
Question q8
===========
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_1.test
***     pacman layout:          Test 1
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_10.test
***     pacman layout:          Test 10
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_11.test
***     pacman layout:          Test 11
***     solution length:                2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_12.test
***     pacman layout:          Test 12
***     solution length:                3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_13.test
***     pacman layout:          Test 13
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_2.test
***     pacman layout:          Test 2
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_3.test
***     pacman layout:          Test 3
***     solution length:                1
***     pacman layout:          Test 4
***     solution length:                3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_5.test
***     pacman layout:          Test 5
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_6.test
***     pacman layout:          Test 6
***     solution length:                2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_7.test
***     pacman layout:          Test 7
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_8.test
***     pacman layout:          Test 8
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_9.test
***     pacman layout:          Test 9
***     solution length:                1

### Question q8: 3/3 ###
```

5. 总结与分析

在本次的 Project1 中，我主要探究了深度优先搜索（DFS）、广度优先搜索（BFS）、一致代价搜索（UCS）和 A*搜索在 Pacman 智能体寻路时的各自表现，并对比了他们之间在各方面的优劣。在完成本次 Project 之前，我对搜索算法还停留在比较基础的理解层面，即在类似问题 1 到问题 4 之类的地形内寻找到从某点到另一点的路径；通过人工智能课程的学习（第 6 周作业中阅读了 A*算法的相关文献，了解到了其在翻译等领域的广泛应用）以及本 Project 中的代码书写（可以自行定义状态信息，而不局限于 Pacman 在地图中的坐标），我对搜索算法的使用与理解都更进了一步，可以说是受益匪浅。下面我将分别对本次 Project 中使用到的四种算法进行理解、优缺点评价、适用场景的介绍。

5.1. 对深度优先搜索（DFS）的总结分析

深度优先搜索从初始状态出发，一直深入到图中没有未被访问的结点为止。如果找到了目标状态，则停止搜索，否则继续回溯。

DFS 的优点是实现简单，并且由于其回溯的特点空间复杂度比较低。它的缺点在于搜索路径可能非常深，会一直沿着一条路径往下走，直到所有路径都走过了才会停止。需要注意的是，必须记录访问过的结点以避免 DFS 在有环路的图中陷入死循环，这也是我起初犯错误的地方。DFS 与后三个算法的最重要区别在于，其不能保证找到最优解。

对于搜索深度较小的问题，深度优先搜索的搜索效率较高，在不要求解的最优性是可以考虑；但是，对于搜索深度比较大的问题或对解的最优性有所要求的问题，DFS 不仅效率有所降低且无法保证解的最优性，因此应该考虑后文的三种算法。

5.2. 对广度优先搜索（BFS）的总结分析

广度优先搜索从初始状态出发，先访问与初始状态相邻的所有结点，然后访问与这些结点相邻的结点，以此类推直到找到目标状态。

BFS 的优点是可以保证找到最短路径，因为它是按照距离由近及远的顺序进行搜索的。但是 BFS 的缺点是空间开销较大，因为它需要存储所有的未被访问的结点，而且需要记录结点之间的关系，这会占用较多的内存空间。需要注意的是，BFS 的时间复杂度为 $O(b^d)$，其中 b 为分支因子，d 为目标结点深度，因此如果目标结点深度非常大或分支因子非常大，BFS 的搜索时间可能会非常长。此外，BFS 只能保证搜索到的结点是无权图中路径最短的（在搜索树中深度最小的），但实际情况中可能结点与结点之间的转移代价不同，因此对于每一步代价不同的场景（图的边有边权）中，BFS 也无法保证搜索到的目标状态是最优状态，需要使用下文提到的 UCS 来保证最优解。

BFS 适用于在每一步代价一致的情况下需要找到最短路径的问题（如问题 8）。

5.3. 对一致代价搜索（UCS）的总结分析

UCS 算法是一种广度优先搜索的变体，它在搜索时考虑了每个结点到起点的距离，选

择距离更短的结点先进行扩展。具体地说，UCS 算法使用优先队列来存储待扩展的结点，并按照结点到起点的距离排序。每次从队列中选取距离最短的结点进行扩展，直到找到目标结点或者队列为空。与 BFS 算法不同，UCS 算法并不仅仅考虑了结点间的距离，而是同时考虑了已经走过的路径长度。

UCS 算法的优点是可以找到代价最小的路径，而非 BFS 中最短的路径（每一步代价相同）。其算法缺点是需要用到优先队列的数据结构不断对所有结点排序，因此空间和时间复杂度都较高。对于搜索空间较大的问题，UCS 算法可能会耗费较长时间。

UCS 算法适用于需要寻找代价最小路径的场景。对于需要保证搜索结果准确性且搜索空间较小的情况，UCS 比较适用。

5.4. 对 A*搜索的总结分析

A*算法是一种启发式搜索算法，它在 UCS 的基础上引入了启发式函数来评估每个结点的价值。它采用估价函数 f(n) = g(n) + h(n)，既考虑了累计代价和，又引入了启发式函数来减少对不必要结点的探索。

A*算法在搜索时能够充分利用启发式函数的信息从而避免无效的搜索，比起 BFS、UCS 能够更快地找到最优解。然而，A*算法的启发式函数需要满足一致性的要求，否则会导致算法搜索到错误的解决方案。在本次 Project 的问题 6、问题 7 中，我着重探究了启发式函数的选择对 A*算法效率的影响。起初找到的一些算法尽管效率不错，但都不满足一致性的前提导致不能找到最优的路径。此后，我又尝试了多种启发式函数的选择，需要在保证其一致性的前提下尽可能的减少探索的结点数，从而提高 A*算法的效率。但在上文的实验结果展示部分，我也列出了两种不同启发式函数对于 A*算法拓展结点数的影响。

与 BFS、UCS 的场景类似，A*算法适用于需要找到最短路径的问题。但其优点在于，A*算法可以通过启发式函数估算每个结点到目标结点的距离，从而更快地找到最短路径。

在完成本次 Project1 实验的过程中，作为人工智能初学者的我深入了解了搜索算法的基本原理和实现方式，并通过 Python 编程实践了这些算法，在不断的调试与改进中对于各个搜索算法的理解与掌握都有了显著的提升。通过撰写与探究深度优先搜索、广度优先搜索、一致代价搜索和 A*搜索算法在不同问题实例中的表现，我对搜索算法的优点、局限性以及适用场景有了更深入的了解，从而可以更好地应用它们来解决实际问题。

此外，通过这个项目，我也培养了解决问题的能力。我需要根据问题要求选择合适的算法，编写代码并测试其正确性和效率。不管是对算法本身的理解、撰写，还是对 Python 语法的学习与熟悉，不断撰写、纠错、调试的过程都让我逐渐提高了自己的分析问题、寻找解决方案和调试代码的能力，以及在解决问题时需要的耐心和毅力。

总的来说，本次 Project1 中我初步探究了人工智能领域的搜索算法内容，也对未来的学习更充满憧憬了。希望本次 Project1 的实验能够为我未来在人工智能领域的学习和研究奠定坚实的基础，也希望自己在未来能够通过更多的学习与实践不断进步！