报告题目：Pacman Game

王崧睿 221502011 detect0530@gmail.com

1 引言

在个人过去的实践中，搜索算法是低效暴力的代名词，但是通过本课程的学习，从深度优先宽度优先到代价优先，再到A*算法，我才发现搜索算法的强大之处。优秀的启发式函数可以大大提高搜索效率，搜索算法的强大之处在于其可以解决各式各样的问题，比如本次实验中的pacman游戏，可以通过搜索算法来解决。并在一次次优化算法的过程中，我也对搜索算法有了更深的理解。

2 实验内容

2.1 TASK1 dfs&bfs in Maze Problem

2.1.1 防止走同样的点

和所有的搜索算法一样，如果遇到了重复的点，那么大可不必再走一遍。

于是在后续所有的程序里，我用$VisitedNode$作为list存储当前的已经走过的状态，如果当前状态已经走过，那么就不再走这个点。

2.1.2 数据结构的选择

在这个实验中，我选择了$Stack$作为深度优先搜索的数据结构，$Queue$作为宽度优先搜索的数据结构。

2.1.3 核心代码展示

```
**宽度优先搜索**

startNode = problem.getStartState()

if problem.isGoalState(startNode):
    return []

Que= util.Queue()
VisitedNode = []
Que.push((startNode, []))

while not Que.isEmpty():
    Nownode, actions = Que.pop()
    if (Nownode not in VisitedNode):
        VisitedNode.append(Nownode)
        if problem.isGoalState(Nownode):
            return actions
        for nextNode, nextAction, cost in problem.getSuccessors(Nownode):
```

```
            newAction = actions + [nextAction]
            Que.push((nextNode, newAction))

    util.raiseNotDefined()
```

2.1.4 实验结果

我们统一用拓展节点数作为评价指标，下面是实验结果：

问题	深度优先搜索	宽度优先搜索
bigMaze	390	620

这是在正确路径只有一条的时候，如果有多条路径，可想而知其expandnode会大大增加。

2.2 TASK2 A* Search in Maze Problem

2.2.1 设计代价函数

在这个实验中，我选择了曼哈顿距离作为启发式函数，即$|x1-x2|+|y1-y2|$。

因为只有一个目标，所以我们的大方向是往$(1,1)$，故而我们的启发式函数是$|x1-1|+|y1-1|$。这样能让我们的程序尽可能往目标点靠近。

(尽管在一些时候，迷宫经过设计，不一定走得通的路径，但是我们的启发式函数能够保证我们的程序能够尽可能的往目标点靠近，而不是南辕北辙乱走。)

```
def myHeuristic(state, problem=None, Map=None):
    (Goal_x, Goal_y) = problem.goal
    return abs(state[0] - Goal_x) + abs(state[1] - Goal_y) + Map[state[0]]
[state[1]]
```

2.2.2 一点优化

对于迷宫问题，一个常见的问题是如何避免走进迷宫的死胡同里。在网格迷宫里，死胡同一定有的特征是最终会到一个三面都是墙的情况。

那么显然，如果我们能预处理判断出那些路是死胡同，那么我们就可以在搜索的时候避免走进死胡同。从而节省大量的$expandNode$开销。

- **2.2.2.1 如果判断死胡同**

所有三面是墙的点都是不必要访问点，其次，如果上下左右有两面墙，且可以走的两个格子里有一个是不必要访问点，那么这个点也是不必要访问点。

- **2.2.2.2 预处理死胡同**

具体地，我们在代码里用map list记录一个格子周围的墙壁数量，如果是3，那么这个点就是死胡同（不必要访问点）。

接着我们进行循环，每一次找map值为2的格子，如果其相邻的格子有不可访问点，那么这个格子也是不可访问点。我们一直执行这个循环直到没有新的不可访问点出现。

```
def Build_Map_With_Block(problem):
    map = [[0 for i in range(problem.walls.height)] for j in
range(problem.walls.width)]
    for i in range(problem.walls.width):
        for j in range(problem.walls.height):
            if problem.walls[i][j]:
                continue
            # print(i,j)
            for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:
                x, y = i, j
                dx, dy = Actions.directionToVector(action)
                nextx, nexty = int(x + dx), int(y + dy)
                if nextx < 0 or nexty < 0 or nextx >= problem.walls.width or nexty
>= problem.walls.height:
                    map[i][j] += 1
                elif problem.walls[nextx][nexty]:
                    map[i][j] += 1
    map[1][1]=0
    while True:
        tot=0
        for i in range(problem.walls.width):
            for j in range(problem.walls.height):
                if problem.walls[i][j]:
                    continue
                for action in [Directions.NORTH, Directions.SOUTH,
Directions.EAST, Directions.WEST]:
                    x, y = i, j
                    dx, dy = Actions.directionToVector(action)
                    nextx, nexty = int(x + dx), int(y + dy)
                    if nextx < 0 or nexty < 0 or nextx >= problem.walls.width or
nexty >= problem.walls.height:
                        continue
                    if map[nextx][nexty] == 3 and map[i][j] == 2:
                        map[i][j] = 3
                        tot+=1
        if tot == 0:
            break
    for i in range(problem.walls.width):
        for j in range(problem.walls.height):
            if map[i][j] == 3:
                map[i][j] = 100000
            elif map[i][j] == 2:
                map[i][j] = 0
            elif map[i][j] == 1:
                map[i][j] = 0
```

```
            elif map[i][j] == 0:
                map[i][j] = 0
    return map
```

可以从代码中看出，我讲不必要走的点map值映射为100000，这样在启发式函数中，加上当前位置的map值，如果是死胡同，那么则不会优先选择这个点。

2.2.4 启发式合并框架

其实和宽度优先搜索很相似，只不过在实现中讲$queue$换成了$priorityqueue$，将$priority$设置为$cost + heuristic$。

2.2.5 核心代码展示

```
def aStarSearch(problem, heuristic=nullHeuristic):

    if heuristic != myHeuristic:
        Map = Build(problem)
    else:
        Map=Build_Map_With_Block(problem)

    startNode = problem.getStartState()

    startcost = heuristic(startNode, problem , Map)
    if problem.isGoalState(startNode):
        return []

    P_Que = util.PriorityQueue()
    VisitedNode = []

    print(startcost)

    # item: (node, action, cost)
    P_Que.push((startNode, [], 0), startcost)
    Show=0
    while not P_Que.isEmpty():
        (currentNode, actions, preCost) = P_Que.pop()

        if Show < preCost + heuristic(currentNode, problem,Map):
            Show = preCost + heuristic(currentNode, problem,Map)
            print("noedepth: {0}".format(Show))

        if (currentNode not in VisitedNode):
            VisitedNode.append(currentNode)

            if problem.isGoalState(currentNode):
                return actions

            for nextNode, nextAction, nextCost in
problem.getSuccessors(currentNode):
                newAction = actions + [nextAction]
```

```
                G_Cost = problem.getCostOfActions(newAction)
                newPriority = G_Cost + heuristic(nextNode, problem,Map)
                P_Que.push((nextNode, newAction, G_Cost), newPriority)

    util.raiseNotDefined()
```

2.2.6 实验结果

我们统一用拓展节点数作为评价指标，下面是实验结果：

问题	A* Search
bigMaze（不加优化）	549
bigMaze（加优化）	266

可以看到，不加优化的A*算法比宽度优先搜索更加优秀。

而在添加优化后，甚至在$bigMaze$这张只有一条路的地图中比dfs还要优秀得多。

证明程序表现相当不错。

2.3 TASK3 A* Search in FoodSearch Problem

2.3.1 问题分析

这是一个相当困难的问题，因为我们的目标是吃到所有的食物，而不是吃到一个食物且要保证最短路径。

我尝试用默认的$nullHeuristic$函数，**但是显然在任何地图中都无法在跑出结果（expandnode根本就不在能跑的数量级上）**。

这意味着我们的启发式函数的设计至关重要。

而如何设计出优秀的启发式函数且满足$admissiable$和$consistant$的要求，且看接下来的分解。

2.3.2 启发式函数的设计

- **Heuristic Function 1**

一种合理的FoodSearch方法是从边缘开始，不断减少由food行成的凸包的直径。

这里的凸包定义为：$convexhull(food)$，代表包住所有食物的最小凸多边形。

而凸包的直径定义为：$max_{x,y\in convexhull(food)}|x-y|$。（由于是在网格上这里的距离为曼哈顿距离）

也就是说我们希望在搜索的同时能够减少凸包的直径，这是符合直觉的。

同时由于我们随机游走也可能导致凸包直径不变，搜索我们还需要引导其去吃豆子，所有增加一个$Mindis$表示当前状态到最近的豆子的距离。

于是我们的启发式函数设计为：$max_{x,y\in convexhull(food)}|x-y|$ + $Mindis$。

- **Heuristic Function 1 的最优性验证**

$Admissible$验证：显然，我们至少需要先走到豆子，然后至少走凸包直径那么多点才能吃到所有的豆子，所以启发式函数一定是小于等于实际距离。

$Consistent$验证：分两种情况：

1. 当前一步没吃到豆子，那么$Mindis$最多减一，而$max_{x,y\in convexhull(food)}|x-y|$不变，所以评价函数的差不会大于1。
2. 当前吃到了豆子，那么$Mindis$不可能比1再小了，同时二维平面上的三角距离不等式知：凸包直径减少量一定比$Mindis$大，所以评价函数的差不会大于1。

综上所述，这个启发式函数是$admissible$和$consistant$的。

- **Heuristic Function 2**

另一种合理的FoodSearch方法是我们尝试让所有点与当前位置计算出$MazeDistance$（即迷宫中的最短距离），然后选择其中的最大值作为启发式函数。这种方法引导我们去不断减小这个最大值，直到所有的豆子都被吃掉。

- **Heuristic Function 2 的最优性验证**

$Admissible$验证：显然，吃完所有豆子至少要走$MaxMazeDistance$步数

$Consistent$验证：走一步$MaxMazeDistance$最多减少一点，所以评价函数的差不会大于1。

Heuristic Function 3

最后针对一字排开引导我们去吃的地图，一种思路是我们尽可能吃能吃的。

对应的启发式函数设计：$Mindis+RemainNumberFood-1$ （Mindis指当前位置到最近的豆子的距离，RemainNumberFood指当前剩余的豆子数量）

- **Heuristic Function 3 的最优性验证**

$Admissible$验证：显然，完成路径至少要走到豆子，且至少还要走完剩下的豆子-1步才能完成。

$Consistent$验证：分两种情况：

1. 没吃到豆子：$Mindis$最多减一，$RemainNumberFood$不变，所以评价函数的差不会大于1。
2. 吃到了豆子：$Mindis$不可能比1再小了，同时$RemainNumberFood$最多减一，所以评价函数的差不会大于1。

所以综上，我们设计了三个启发式函数，为了最优化我们取

$$HeuristicFuction()=max(H1(),H2(),H3())$$

作为最终的启发式函数。

2.3.3 计算Heuristic Function的准备工作

- H1

计算凸包，尽管可以使用更优秀的算法在$O(nlogn)$的时间复杂度内计算凸包直径，但是在本题中凸包的计算并不是代码瓶颈，故而采用暴力枚举所有food点对进行计算。

```
def H1(state,problem):
    position, foodGrid = state

    Maxdis = 0
    Mindis = 99999
    x1, y1 = 0, 0
    x2, y2 = 0, 0
    Sum = 0
    for i, j in foodGrid.asList():
        Sum += abs(i - position[0]) + abs(j - position[1])
        for k, l in foodGrid.asList():
            if (abs(i - k) + abs(j - l) >= Maxdis):
                x1, y1 = i, j
                x2, y2 = k, l
                Maxdis = abs(i - k) + abs(j - l)

    Mindis = min(abs(x1 - position[0]) + abs(y1 - position[1]), abs(x2 -
position[0]) + abs(y2 - position[1]))

    if foodGrid.count() == 0:
        return 0

    return Maxdis + Mindis
```

- H2

实现了$MazeDistance$的计算，这里预先使用$dijkstra$算法将所有地图点对的最短距离存储在了一个四维数组中，而在查询时可以直接调用数组，从而可以大大加快计算速度。

```
def Dij(S,T,problem):

    """ 计算S位置与T位置的迷宫最短留"""

    P_Que = util.PriorityQueue()
    VisitedNode = []
    P_Que.push((S,0) , 0)
    while not P_Que.isEmpty():
        Now = P_Que.pop()
        cur, dis = Now[0], Now[1]
     #   print(cur,"##",dis)
        if cur not in VisitedNode:
            VisitedNode.append(cur)
            if cur == T: return dis
```

```
            for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:
                x, y = cur[0],cur[1]
                dx, dy = Actions.directionToVector(action)
                nextx, nexty = int(x + dx), int(y + dy)
                if nextx < 0 or nexty < 0 or nextx >= problem.walls.width or nexty
>= problem.walls.height or problem.walls[nextx][nexty] or
VisitedNode.__contains__((nextx,nexty)):
                    continue
                else:
                    P_Que.push(((nextx,nexty),dis+1),dis+1)

def Build(problem):
    A = C = problem.walls.width
    B = D = problem.walls.height
    Map = [[[[(0) for _ in range(D)] for _ in range(C)] for _ in range(B)] for _
in range(A)]

    for i in range(problem.walls.width):
        for j in range(problem.walls.height):
            for k in range(problem.walls.width):
                for l in range(problem.walls.height):
                    if problem.walls[i][j] or problem.walls[k][l]:
                        continue
                    Map[i][j][k][l] = Dij((i,j),(k,l),problem)
    return Map
```

- H3

没有什么细节，直接调用$foodGrid.count()$即可。

```
def H3(state,problem):
    position, foodGrid = state

    Mindis = 99999
    for i, j in foodGrid.asList():
        Mindis = min(Mindis,abs(i - position[0]) + abs(j - position[1]))

    if foodGrid.count() == 0:
        return 0

    return Mindis + foodGrid.count()-1
```

2.3.4 核心框架

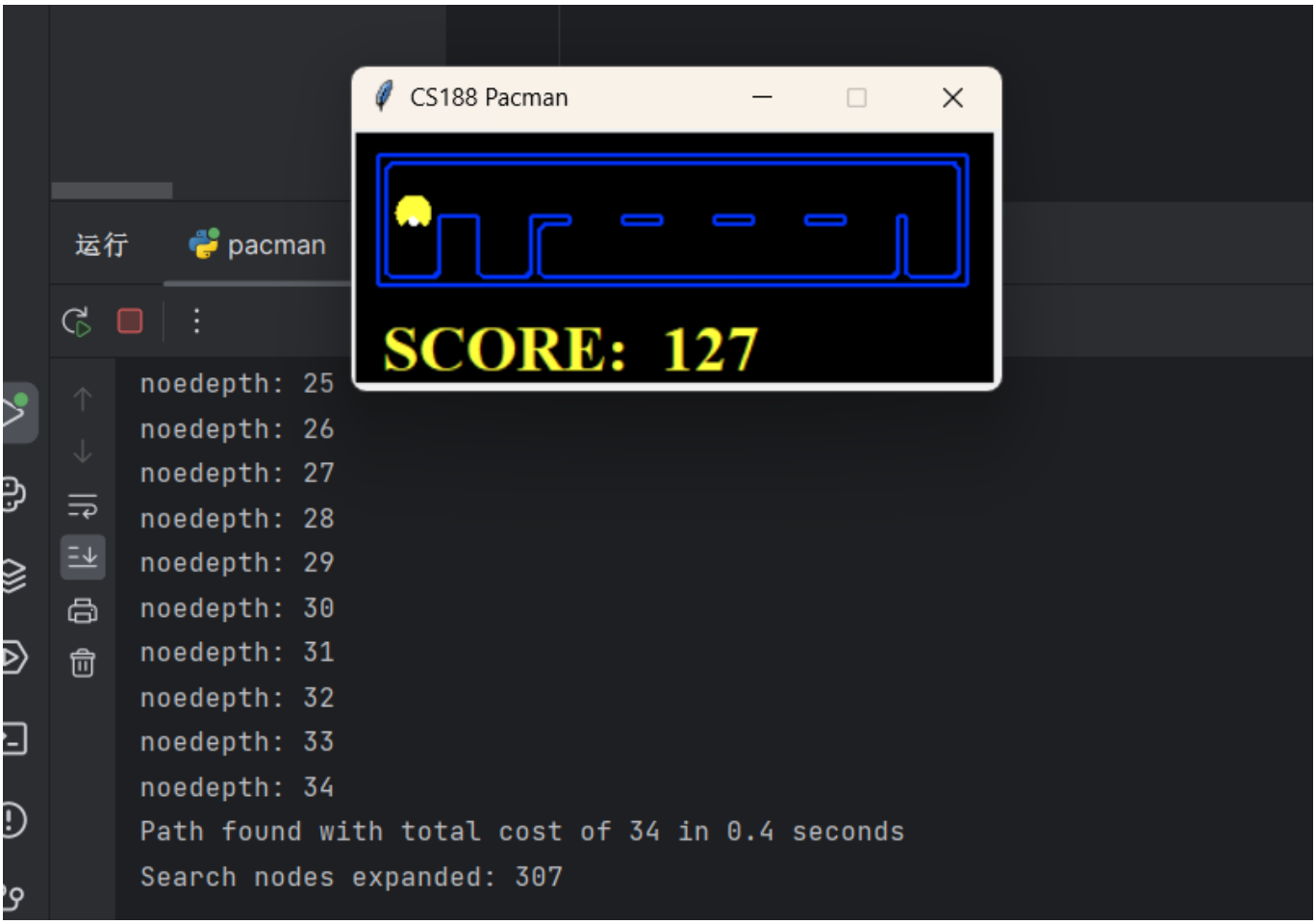
和TASK2中的A*算法框架相同，只不过将启发式函数换成了$HeuristicFuction() = max(H1(), H2(), H3())$。

甚至就是调佣的同一个函数，只是改了启发式函数调用的指针而已。

2.3.5 实验结果

我们统一用拓展节点数作为评价指标，下面是实验结果：

问题	A* Search	nullHeuristic
Search	31	∞
smallSearch	307	∞



可以看到，我们的启发式函数在这两个问题上表现相当优秀。

（吐槽剩下的两个图太大了，这本来就是np问题，地图太大没办法跑也就没法继续评估A*，但是可以看到在两个能跑的问题上，我的A*能跑出相当优秀的结果）

---

3 总结

在本次作业中，我分别实现了深度优先搜索，宽度优先，Astar。

从结果上来看，发现Astar算法的效果最好，而且在不同的地图中，只需要调整一下权值就可以很好的适应不同的地图。

并且通过本次作业的学习，我对搜索算法有了不一样的认识，也对搜索算法的实现有了更深的理解。（debug痛苦呜呜呜

以上。