人工智能实验报告

--project1 search

学院：计算机科学与技术学院

班级：1403202

学号：1140320206

姓名：霍峻杰

1. 简介

在人工智能盛行的大环境下，各种自动运行的程序，机器使生活变得非常的简单易行。人工智能中的agent代名词也越来越被人们重视起来。而人工智能的的前提是，机器或程序可以自动寻找到最优的办法或方式来完成某种功能或目的。

本实验，基于路径搜索问题模拟人工智能，包括自动寻路，自动寻找吃掉所有可吃点的最短路径，以及启发式的思想。本实验的目的是通过模拟简单的人工智能搜索路径来使我们初步接触人工智能，了解人工智能的具体用途，将理论知识付诸于实践，在实践中理解和提高。

1. 方法
2. 应用深度优先算法找到一个特定的位置的豆
   1. 分析：

首先，分析深度优先，宽度优先，代价一致性，A\*算法，这几种算法的结构其实都是一样的，不同的地方在于数据结构的选择是不同的。深度优先用到的数据结构是栈结构(后进先出)，宽度优先用到的数据结构是队列，代价一致性用到的数据结构是优先队列，而A\*算法用到的数据结构也是优先队列，他和代价一致性的区别在于多了一个启发式的预测代价h，相当于对代价一致性算法的改进和优化。所以，对图的整体路径算法思想直接在第一部分给出，问题2，3，4简略介绍。

从起始位置开始，找到可到达的点，就压入数据结构中，准备进行下一次终结状态判断，如果遇到终止状态，则已经找到搜索路径，返回即可。

* 1. 解决方法：

下面给出基本的图最短路径搜索算法:

* + 1. 定义一个visited列表，用于记录被访问过的节点
    2. 将初始状态三元组压入数据结构（三元组分别记录状态，方向，访问步数）
    3. 循环弹出数据结构中的元素，判断是否为终止状态
    4. 如不是终止状态，找到未访问的下一个可达点，将其压入数据结构中。

数据结构选择栈，栈是先进后出原则，可以达到深度优先搜索的目的。

1. 宽度优先算法
   1. 分析：

图的搜索思想及算法已经在问题1中给出，不再赘述。这里用到的是队列，遵循先进先出的原则，可以达到广度优先的目的。

* 1. 解决方法：

上面已经给出基本的图最短路径搜索算法。数据结构选择队列，队列是先进先出原则，可以达到宽度优先搜索的目的。

1. 宽度优先算法（代价一致性）
   1. 分析：

图的搜索思想及算法已经在问题1中给出，不再赘述。这里用到的是优先队列，遵循先进，按优先级出的原则，其中，代价cost作为优先级，可以达到代价一致搜索的目的。

* 1. 解决方法：

上面已经给出基本的图最短路径搜索算法。数据结构选择优先队列，优先队列是先进，按优先级出原则，可以达到代价一致搜索的目的。

1. A\* 算法
   1. 分析：

图的搜索思想及算法已经在问题1中给出，不再赘述。这里用到的是优先队列。基本思想和代价一致算法相似，优化的地方在于多了对后续结果的预测值h，可以达到A\*搜索的目的。

* 1. 解决方法：

上面已经给出基本的图最短路径搜索算法。数据结构选择优先队列，优先队列是先进，按优先级出原则，代价cost加预测值h，作为优先级，可以达到A\*搜索的目的。

1. 找到所有的角落
   1. 分析：

因为问题的最终目的是将四个角落的点都吃完，所以当未吃点列表为空时，算法结束。初始位置定义当前点的位置和地图上四个角落的点的坐标。找一个点的下一状态时，需要上下左右进行判断是否撞墙，如果不撞墙，加入到可达下一状态列表中。然后调用bfs算法进行路径搜索。

* 1. 解决方法：
     1. 定义初始状态，点的起始出发位置坐标和四个角落点的坐标。
     2. 定义终止状态，当未吃点列表为空（吃完最后一个豆子）时，立即结束。
     3. 定义获得当前状态的下一可达子状态，分别尝试从当前点到上下左右四个状态点是否撞墙，如果不撞墙，则加入到可达子状态及其方向，步数组成的三元组。如果走到的点是角落里的未吃点，则在未吃点列表中将刚才吃掉的点删除。
     4. Bfs遍历找出吃到所有点的最短路径（用到问题2的bfs）

1. 角落问题（启发式）
   1. 分析：

吃豆人按照当前点距四个角落的距离选择出，角落中据当前位置最近的角落，首先吃这个点，然后选择距现在位置最近的点，进行路径搜索，知道把所有的点都吃完。

* 1. 解决方法：
     1. 判断当前点的位置距哪个角落最近
     2. 估计值加上当前点到最近角落的距离
     3. 当前点变为刚才选中的最近角落（已经走到了）
     4. 从未点列表中删除刚才已经吃掉的点
     5. 循环I,ii,iii,iv,直到未删除点列表为空

1. 吃掉所有的豆子（启发式）
   1. 分析：

得到未吃点列表，与上述启发式相似，都是找到与当前点距离最近的点作为下一状态点，循环，直到未吃点列表为空。

* 1. 解决方法：
     1. asList得到未吃点列表
     2. 计算出据当前点最近的点
     3. 下一状态即为刚才计算出的最近的点
     4. 将刚才吃掉的点从未吃点列表中删除
     5. 重复i—iv，直到未吃点列表为空

1. 次最优搜索
   1. 分析：

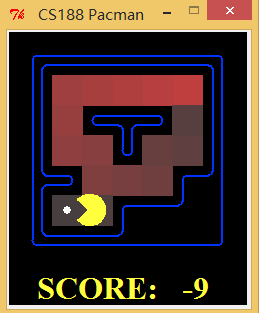
需要改进最近点搜索算法，之前用的循环遍历，效率非常低，可以用A\*算法将最近点找出。定义终结状态，当到达终点时，food[x][y]，终止。

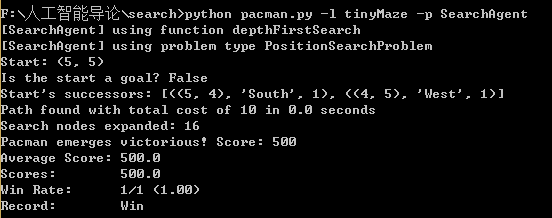
* 1. 解决方法：
     1. 最近点搜索，A\*算法找到距当前点最近的点
     2. 终结状态定义，food[x][y]为终结状态

1. 实验结果
2. 深度优先搜索
   1. 测试命令1

python pacman.py -l tinyMaze -p SearchAgent

测试结果1

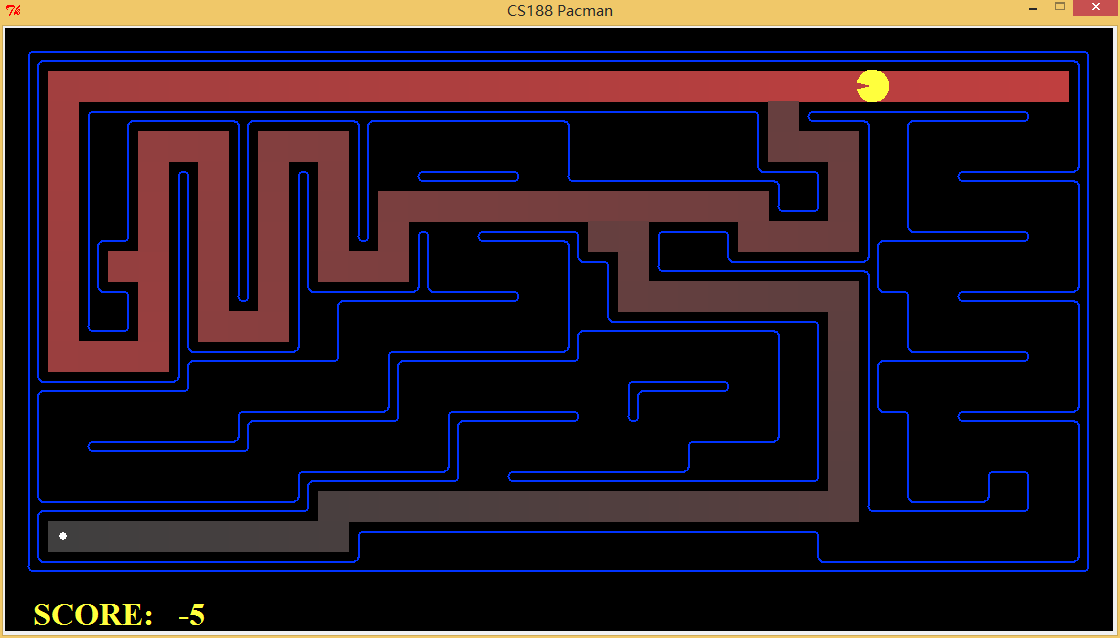


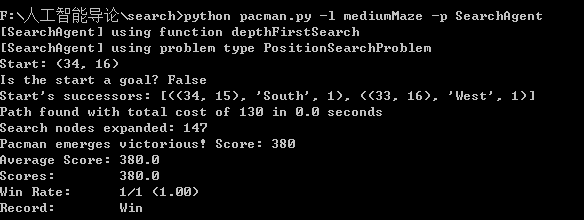


* 1. 测试命令2

python pacman.py -l mediumMaze -p SearchAgent

测试结果2

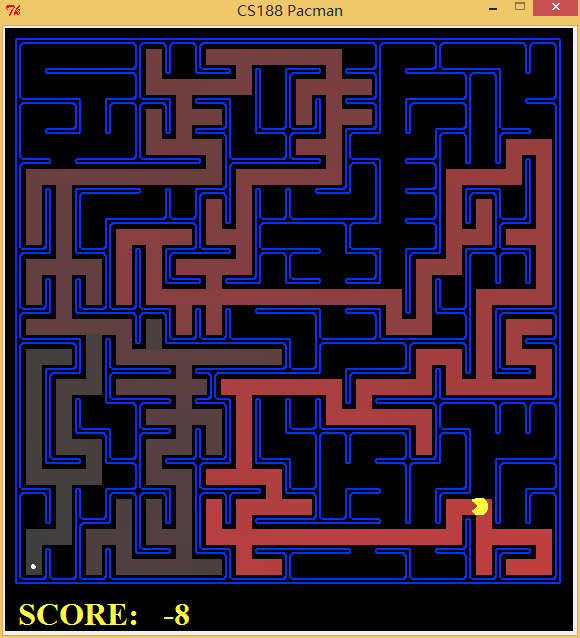


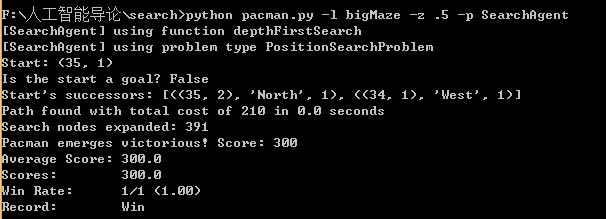


* 1. 测试命令3

python pacman.py -l bigMaze -z .5 -p SearchAgent

测试结果3

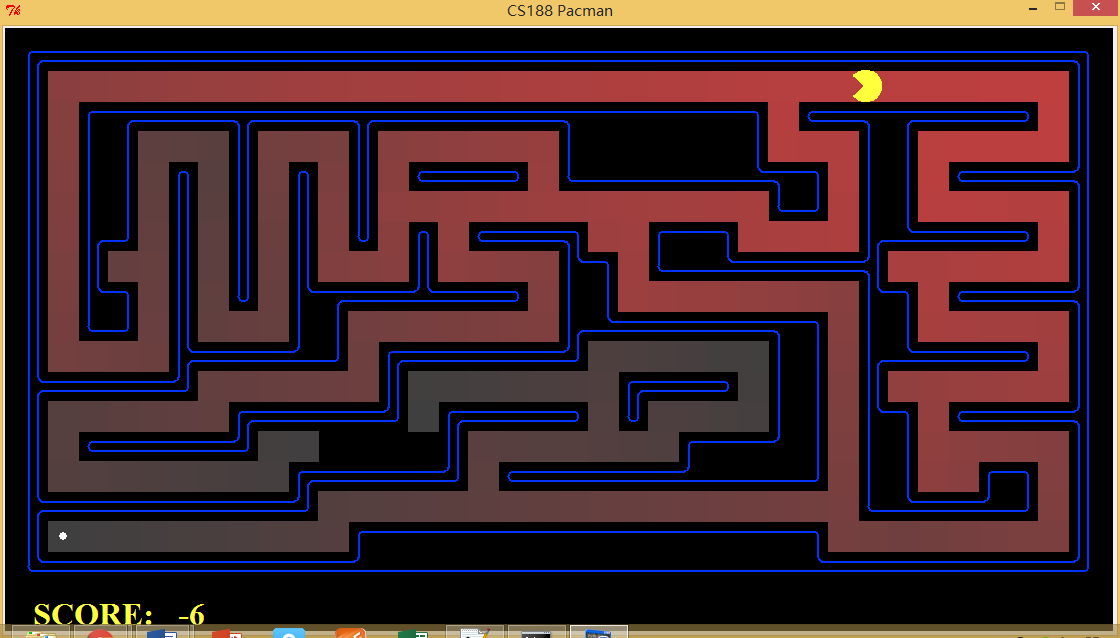


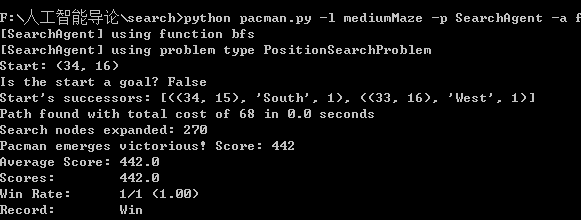


1. 宽度优先搜索
   1. 测试命令1

python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

测试结果1

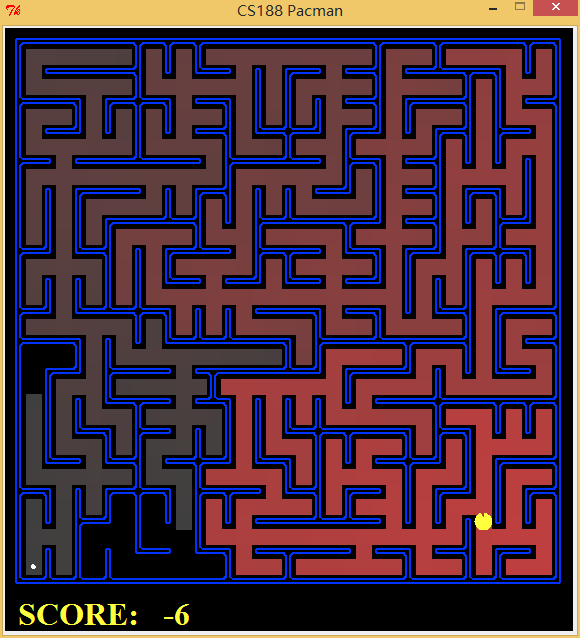


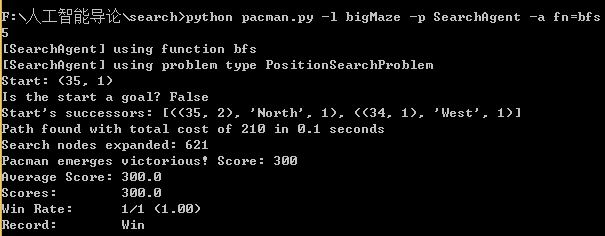


* 1. 测试命令2

python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

测试结果2

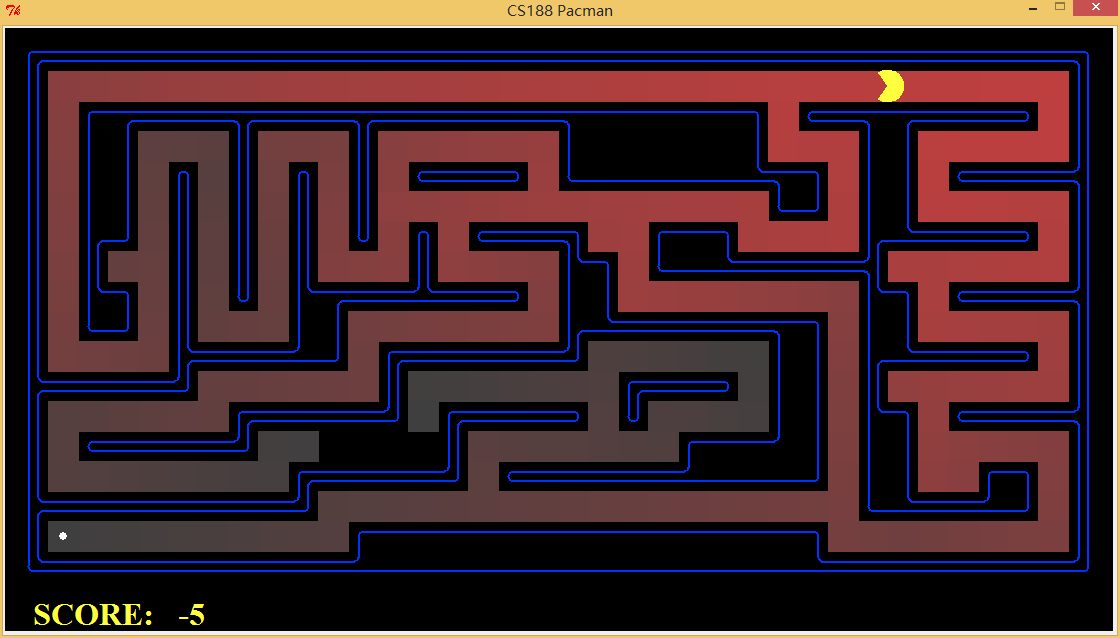


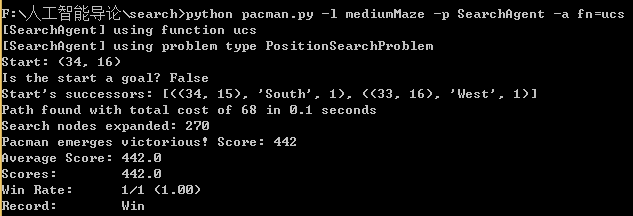


1. 宽度优先搜索（代价一致）
   1. 测试命令1

python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

测试结果1

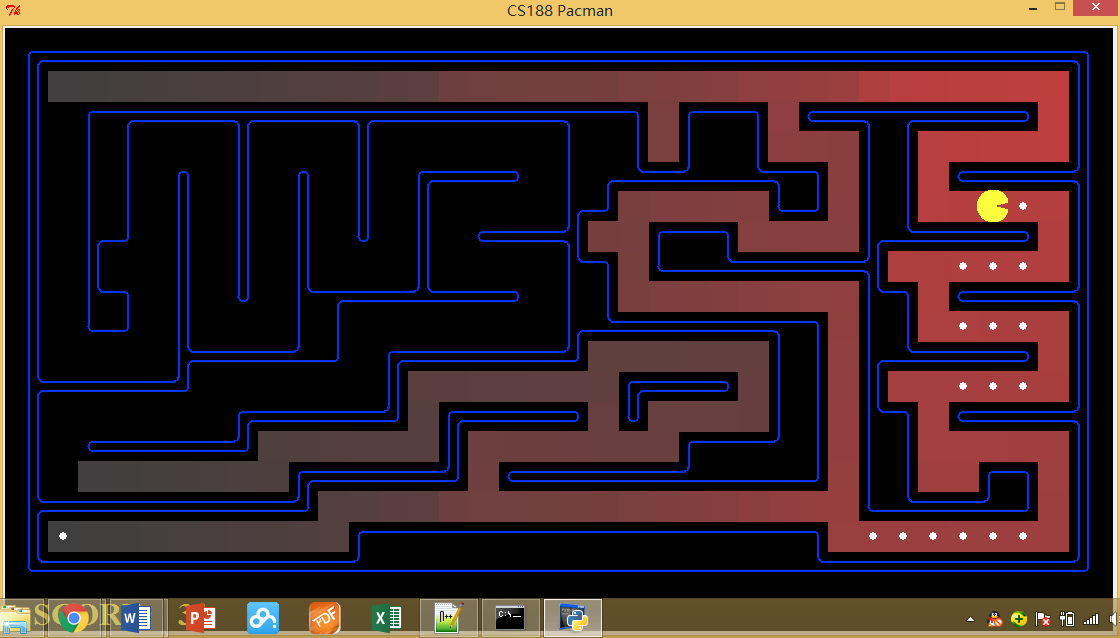


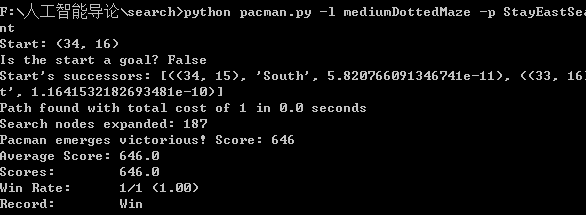


* 1. 测试命令2

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

测试结果2

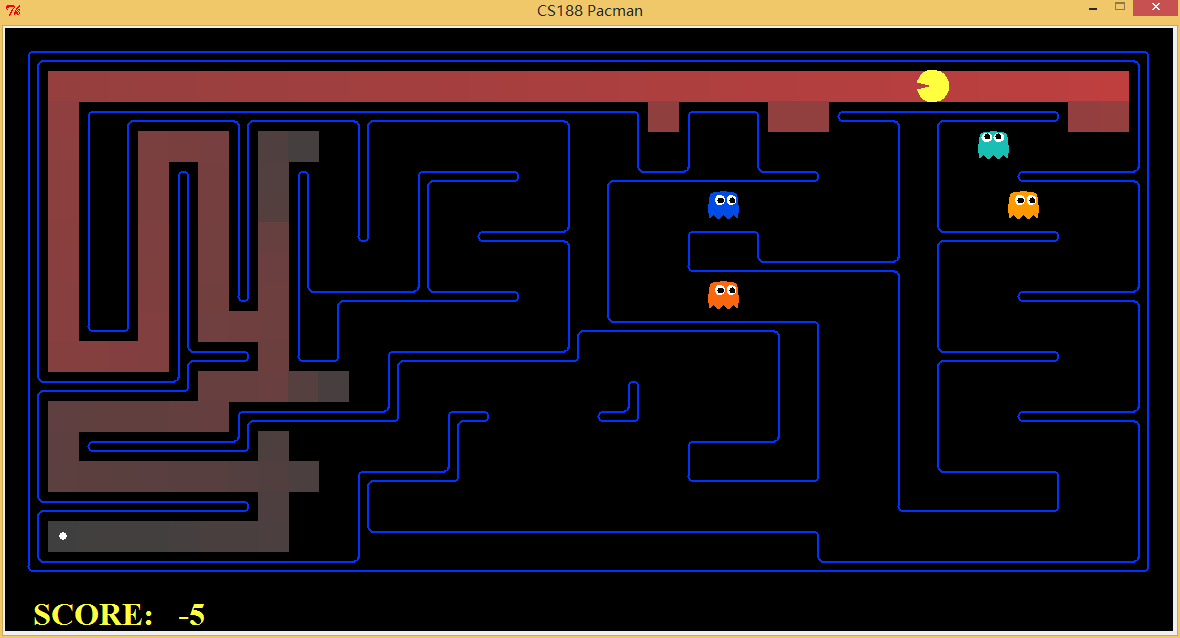


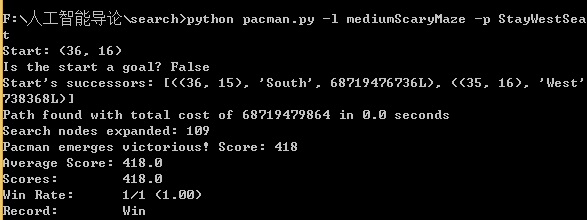


* 1. 测试命令3

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

测试结果3

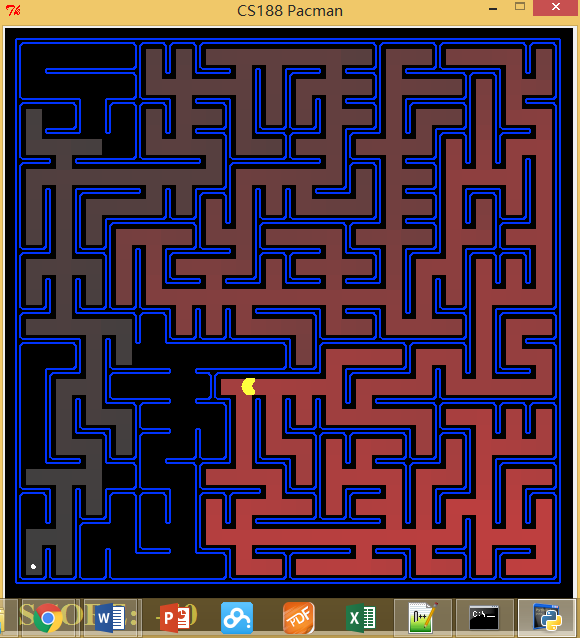


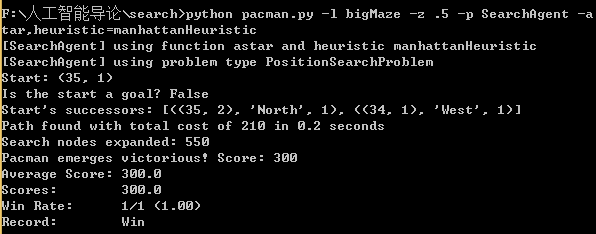


1. A\*算法
   1. 测试命令1

python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

测试结果1

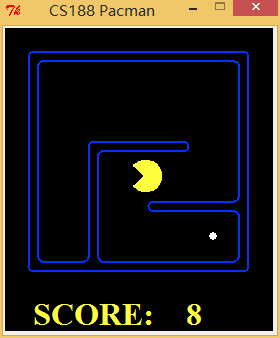


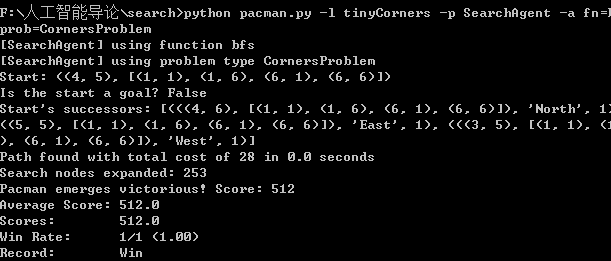


1. 找到所有的角落
   1. 测试命令1

python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

测试结果1

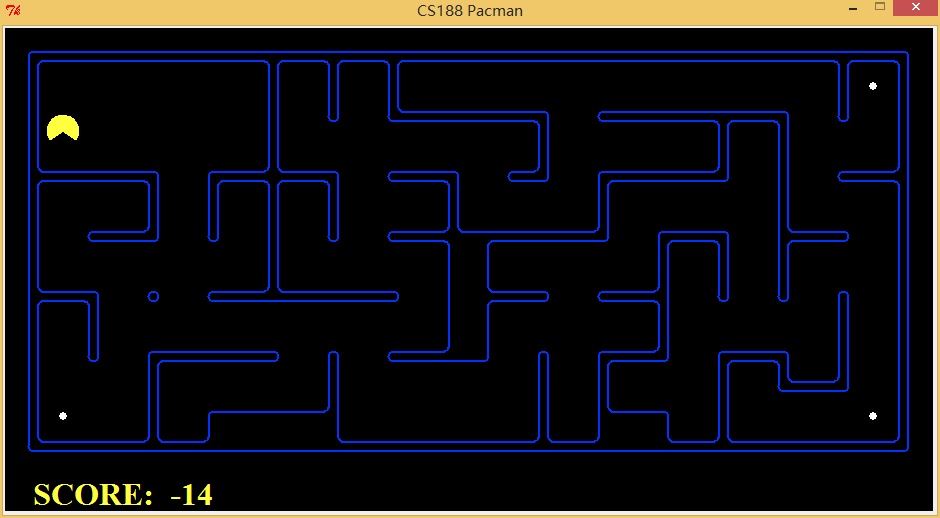


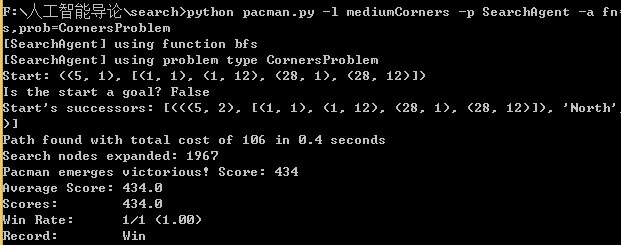


* 1. 测试命令2

python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

测试结果2



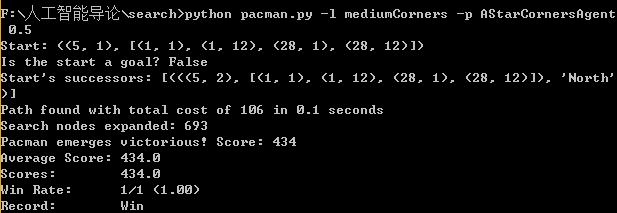


1. 角落问题（启发式）
   1. 测试命令1

python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

测试结果1

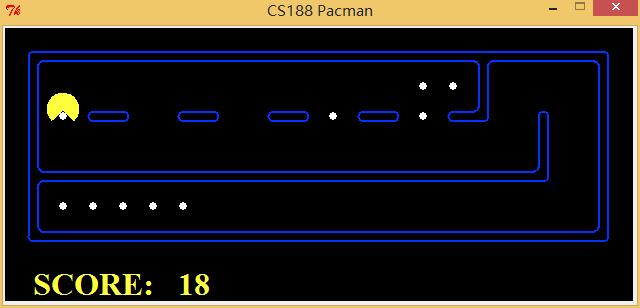


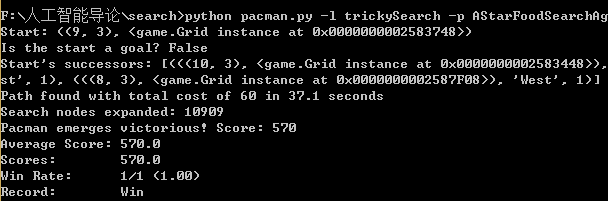


1. 找到所有豆子
   1. 测试命令1

python pacman.py -l trickySearch -p AStarFoodSearchAgent

测试结果1

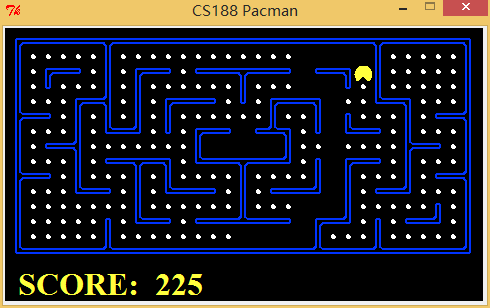


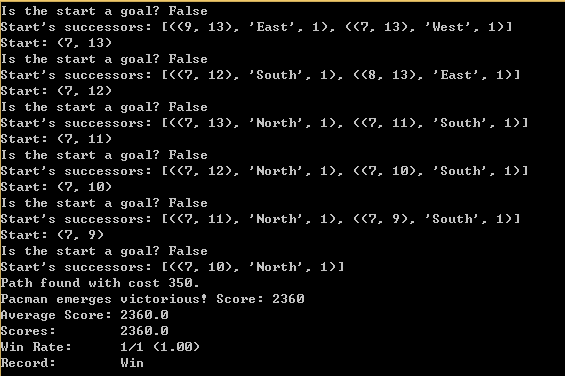


1. 次优先搜索
   1. 测试命令1

python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

测试结果1





1. 总结及讨论

本次实验通过一个吃豆人的小游戏，使我们将一些人工智能搜索路径的方法实践化。同时，更深刻的认识了更重短发之间的联系及算法复杂度。通过对比可知，深度，宽度针对不同地图的到达步数不一定谁长，两者思想不同，复杂度却极为相似。代价一致算法和A\*算法很像，A\*比他多了一个估计值h作为优化，可以认为A\*算法是对代价一致算法的一种改进。

除此之外，我们还接触了吃豆子问题，将问题化解为找到当前位置最近的位置的启发性思想。在进行问题7时，我们发现，在找最近点时，遍历的方法十分的低效，运行时间需要很长，于是，问题8通过A\*算法找到了计算最近点的快速方法，提速很多。

一开始，觉得这次的实验非常的复杂，但在这种有游戏趣味性的实验中，我感到非常的感兴趣，通过一晚上的研究，写完了代码，并加上了详细的注释（逐行注释）。

感觉这次收获非常大，以前一直觉得人工智能虽然可以想象到，但是却离我们比较远，今天发现，原来人工智能的小小应用是这么的有趣。

1. 附录（添加的代码）

如果看不清我代码，请登陆我的blog，lovebear.top

1. 问题1：

def graphSearch(problem, fringe):

pgss=problem.getStartState() #获取problem的初始状态

print "Start:", pgss #打印problem的初始状态

print "Is the start a goal?", problem.isGoalState(pgss) #判断初始状态是目标状态吗

print "Start's successors:", problem.getSuccessors(pgss) #打印初始状态的子状态（与这一状态连接的可达状态）

visited = [] #记录这个状态是否访问过，访问过的状态都被添加到visited列表里

fringe.push([(pgss, "Stop" , 0)]) #向数据及结构（依fringe定义的数据结构而定）压入初始状态三元组构成

#的访问路径，三元组元素分别为状态，方向，访问步数

while not fringe.isEmpty(): #数据结构不为空一直循环

path = fringe.pop() #将数据弹出一个（栈顶，队首...）路径

s = path[len(path)-1] #取得路径末的三元组（及访问路径中最后一个状态所在的三元组）

s = s[0] #取得本路径的最后状态

if problem.isGoalState(s): #如果取得的这最后一个状态就是我们的目标状态，则搜索结束

return [x[1] for x in path][1:] #返回路径中所有的方向（除去初始状态的方向）

if s not in visited: #如果这个状态还没被访问过

visited.append(s) #访问这个状态，并将这个状态添加到访问记录列表visited[]中

for successor in problem.getSuccessors(s): #遍历现状态的子状态

if successor[0] not in visited: #子状态没有被访问过

successorPath = path[:] #创建新的子状态路径（复制当前状态的路径），此处的[:]

#为了防止python的引用处理，这样可以真正创建新的路径

successorPath.append(successor)#在原状态路径后面添加上当前访问的子状态，得到新的状态路径

fringe.push(successorPath) #将新的状态路径压入数据结构中

return []

def depthFirstSearch(problem):

"""

Search the deepest nodes in the search tree first.

Your search algorithm needs to return a list of actions that reaches the

goal. Make sure to implement a graph search algorithm.

To get started, you might want to try some of these simple commands to

understand the search problem that is being passed in:

print "Start:", problem.getStartState()

print "Is the start a goal?", problem.isGoalState(problem.getStartState())

print "Start's successors:", problem.getSuccessors(problem.getStartState())

"""

"\*\*\* YOUR CODE HERE \*\*\*"

fringe = util.Stack() #fringe数据结构为栈，进行dfs

return graphSearch(problem, fringe)

1. 问题2：

def breadthFirstSearch(problem):

"""Search the shallowest nodes in the search tree first."""

"\*\*\* YOUR CODE HERE \*\*\*"

fringe = util.Queue() #fringe数据结构为队列，进行bfs

return graphSearch(problem, fringe)

1. 问题3：

def uniformCostSearch(problem):

"""Search the node of least total cost first."""

"\*\*\* YOUR CODE HERE \*\*\*"

cost = lambda aPath: problem.getCostOfActions([x[1] for x in aPath]) #计算出每条路径的总代价，通过总代价

#作为优先级进行搜索

fringe = util.PriorityQueueWithFunction(cost) #fringe数据结构为优先队列

return graphSearch(problem, fringe)

1. 问题4：

def nullHeuristic(state, problem=None):

"""

A heuristic function estimates the cost from the current state to the nearest

goal in the provided SearchProblem. This heuristic is trivial.

"""

return 0

def aStarSearch(problem, heuristic=nullHeuristic):

"""Search the node that has the lowest combined cost and heuristic first."""

"\*\*\* YOUR CODE HERE \*\*\*"

cost = lambda aPath: problem.getCostOfActions([x[1] for x in aPath]) + heuristic(aPath[len(aPath)-1][0], problem)

fringe = util.PriorityQueueWithFunction(cost) #A\*是对代价一致搜索算法的改进，加入了一个估计代价h

return graphSearch(problem, fringe)

1. 问题5：

def getStartState(self):

"""

Returns the start state (in your state space, not the full Pacman state

space)

"""

"\*\*\* YOUR CODE HERE \*\*\*"

return (self.startingPosition, [self.corners[0], self.corners[1], self.corners[2], self.corners[3]])

#agent初始状态的位置，和四个角落的豆的位置

def isGoalState(self, state):

"""

Returns whether this search state is a goal state of the problem.

"""

"\*\*\* YOUR CODE HERE \*\*\*"

return len(state[1]) == 0 #state[0]为当前agent位置， state[1]为还没吃到的点的位置，state[1]为

#空时，结束为终止位置

def getSuccessors(self, state):

"""

Returns successor states, the actions they require, and a cost of 1.

As noted in search.py:

For a given state, this should return a list of triples, (successor,

action, stepCost), where 'successor' is a successor to the current

state, 'action' is the action required to get there, and 'stepCost'

is the incremental cost of expanding to that successor

"""

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 = currentPosition

# dx, dy = Actions.directionToVector(action)

# nextx, nexty = int(x + dx), int(y + dy)

# hitsWall = self.walls[nextx][nexty]

"\*\*\* YOUR CODE HERE \*\*\*"

x,y = state[0] #获取agent当前位置

dx, dy = Actions.directionToVector(action)

nextx, nexty = int(x + dx), int(y + dy) #下一个位置坐标

hitsWall = self.walls[nextx][nexty] #判断是否撞到墙

if not hitsWall: #判断，没撞到墙

nextPosition = (nextx, nexty) #下个位置坐标

cornersLeft = state[1][:] #复制当前剩余没吃到的点

if nextPosition in cornersLeft: #如果下个坐标在剩余没吃到的点中

cornersLeft.remove(nextPosition) #因为下一个就能把它吃掉，所以把这个点从剩余

#没吃到点的列表中删除

nextState = (nextPosition, cornersLeft) #计算下一个点的坐标

cost = 1 #代价为1，（步数为1）

successors.append( ( nextState, action, cost) ) #属于可达的子状态，加入到子状态列表

self.\_expanded += 1 # DO NOT CHANGE

return successors

1. 问题6：

def cornersHeuristic(state, problem):

"""

A heuristic for the CornersProblem that you defined.

state: The current search state

(a data structure you chose in your search problem)

problem: The CornersProblem instance for this layout.

This function should always return a number that is a lower bound on the

shortest path from the state to a goal of the problem; i.e. it should be

admissible (as well as consistent).

"""

corners = problem.corners # These are the corner coordinates

walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

"\*\*\* YOUR CODE HERE \*\*\*"

heuristic = 0

cornersLeft = state[1][:] #剩余点的坐标

referencePoint = state[0] #当前点的坐标

while len(cornersLeft) > 0: #当有点剩余（未吃完所有点）

closestCorner = closestPoint(referencePoint, cornersLeft) #当前点坐标和未吃到点中的哪一个离得最近

heuristic += euclidieanDistance(referencePoint, closestCorner) #估计值h加上当前点到最近未吃点的距离

referencePoint = closestCorner #当前点变为最近未吃点，因为已经走到了那个点

cornersLeft.remove(closestCorner) #从未吃点列表中将刚才吃掉的点删除

return heuristic

#return 0 # Default to trivial solution

def closestPoint (fromPoint, candidatesList):

if len(candidatesList) == 0: #如果未吃点列表为空，证明所有点都被吃了，返回none

return None

closestCorner = candidatesList[0] #获取未吃点列表中的第一个点

closestCost = euclidieanDistance(fromPoint, closestCorner) #计算当前点到第一个点距离

for candidate in candidatesList[1:]:

thisCost = euclidieanDistance(fromPoint, candidate)

if closestCost > thisCost: #循环遍历列表，返回距当前点最近的一个未吃点的坐标

closestCost = thisCost

closestCorner = candidate

return closestCorner

def euclidieanDistance (pointA, pointB):

return abs(pointA[0] - pointB[0]) + abs(pointA[1] - pointB[1])#因为坐标为表格化，不能斜着走，所以最短距离为|x|+|y|

1. 问题7：

def foodHeuristic(state, problem):

"""

Your heuristic for the FoodSearchProblem goes here.

This heuristic must be consistent to ensure correctness. First, try to come

up with an admissible heuristic; almost all admissible heuristics will be

consistent as well.

If using A\* ever finds a solution that is worse uniform cost search finds,

your heuristic is \*not\* consistent, and probably not admissible! On the

other hand, inadmissible or inconsistent heuristics may find optimal

solutions, so be careful.

The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a Grid

(see game.py) of either True or False. You can call foodGrid.asList() to get

a list of food coordinates instead.

If you want access to info like walls, capsules, etc., you can query the

problem. For example, problem.walls gives you a Grid of where the walls

are.

If you want to \*store\* information to be reused in other calls to the

heuristic, there is a dictionary called problem.heuristicInfo that you can

use. For example, if you only want to count the walls once and store that

value, try: problem.heuristicInfo['wallCount'] = problem.walls.count()

Subsequent calls to this heuristic can access

problem.heuristicInfo['wallCount']

"""

position, foodGrid = state

"\*\*\* YOUR CODE HERE \*\*\*"

foodList = foodGrid.asList() #得到要吃的点的列表（未吃点列表）

if len(foodList) == 0: #当未吃点列表为空时，停止

return 0

closestFood = closestPoint(position, foodList) #得到离当前点最近的未吃食物

return euclidieanDistance(position, closestFood) + len(foodList)

1. 问题8：

class ClosestDotSearchAgent(SearchAgent):

"Search for all food using a sequence of searches"

def registerInitialState(self, state):

self.actions = []

currentState = state

while(currentState.getFood().count() > 0):

nextPathSegment = self.findPathToClosestDot(currentState) # The missing piece

self.actions += nextPathSegment

for action in nextPathSegment:

legal = currentState.getLegalActions()

if action not in legal:

t = (str(action), str(currentState))

raise Exception, 'findPathToClosestDot returned an illegal move: %s!\n%s' % t

currentState = currentState.generateSuccessor(0, action)

self.actionIndex = 0

print 'Path found with cost %d.' % len(self.actions)

def findPathToClosestDot(self, 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 \*\*\*"

return search.aStarSearch(problem,search.nullHeuristic) #A\*搜索

class AnyFoodSearchProblem(PositionSearchProblem):

"""

A search problem for finding a path to any food.

This search problem is just like the PositionSearchProblem, but has a

different goal test, which you need to fill in below. The state space and

successor function do not need to be changed.

The class definition above, AnyFoodSearchProblem(PositionSearchProblem),

inherits the methods of the PositionSearchProblem.

You can use this search problem to help you fill in the findPathToClosestDot

method.

"""

def \_\_init\_\_(self, gameState):

"Stores information from the gameState. You don't need to change this."

# Store the food for later reference

self.food = gameState.getFood()

# Store info for the PositionSearchProblem (no need to change this)

self.walls = gameState.getWalls()

self.startState = gameState.getPacmanPosition()

self.costFn = lambda x: 1

self.\_visited, self.\_visitedlist, self.\_expanded = {}, [], 0 # DO NOT CHANGE

def isGoalState(self, state):

"""

The state is Pacman's position. Fill this in with a goal test that will

complete the problem definition.

"""

x,y = state

"\*\*\* YOUR CODE HERE \*\*\*"

return self.food[x][y] #state为终点坐标

P．S：代码运行结果是正确的，但是在autograder.py中不能给出正确的grade，因为，我们在expanded\_state中多了一个初始状态，导致自动评分系统结果不符，但是运行的结果是正确的。

