实验0：Python学习

1. 实验目的
2. 简短的Unix/Linux基础学习；
3. Python环境安装；
4. 简短的Python基础学习；
5. 实验成绩计算：每一次实验的发布都包括了自动计算成绩，可以自己运行来进行计算。

2、实验设计

填写压缩包tutorial.zip中如下三个文件的相应部分： addition.py、buyLotsOfFruit.py 和shopSmart.py，最终提交包括了代码和注释的这三个文件。

**问题1: Addition**

问题分析：

填写文件addition.py中的函数addition， 输入：a，b；输出：a加b的值；

解决方案：

将a和b相加之后返回；

程序实现：

1. def add(a, b):
2. "Return the sum of a and b"
3. "\*\*\* YOUR CODE HERE \*\*\*"
4. return a + b

**问题2: buyLotsOfFruitfunction**

问题分析：

填写文件buyLotsOfFruit.py中的函数buyLotsOfFruit， 输入：列表(fruit,pound)，输出：列表对象对应的总价；

解决方案：

使用totalCost记录总花费，通过orderList索引fruitPrices并乘以对应数量并相加。

程序实现：

1. def buyLotsOfFruit(orderList):
2. """
3. orderList: List of (fruit, numPounds) tuples
4. Returns cost of order
5. """
6. totalCost = 0.0
7. "\*\*\* YOUR CODE HERE \*\*\*"
8. for order in orderList:
9. totalCost = totalCost + fruitPrices[order[0]]\*order[1]
10. return totalCost

**问题3:shopSmartfunction**

问题分析：

填写文件shopSmart.py中函数shopSmart(orders,shops)， 输入：列表orderList和FruitShop，输出：最小代价的FruitShop。

解决方案：

预设一个很大的花费(best)，并循环比较多个超市花费综合，并使用较小值更新best值。

程序实现：

1. def shopSmart(orderList, fruitShops):
2. """
3. orderList: List of (fruit, numPound) tuples
4. fruitShops: List of FruitShops
5. """
6. "\*\*\* YOUR CODE HERE \*\*\*"
7. bestShop = fruitShops[1]
8. best = 1000
9. for shop in fruitShops:
10. tmp = 0
11. for order in orderList:
12. tmp = tmp + shop.fruitPrices[order[0]]\*order[1]
13. if tmp < best:
14. best = tmp
15. bestShop = shop
16. return bestShop

3、结果分析

**问题1: Addition**

实验验证方法：python autograder.py

实验结果：



结果评价：输入a=1,b=2,输出为3，结果正确，成功实现add函数功能。

**问题2: buyLotsOfFruitfunction**

实验验证方法：python autograder.py

实验结果：

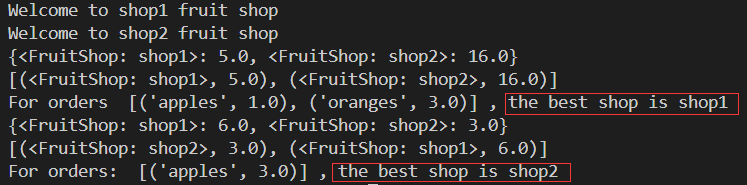


结果评价：输入购买列表orderList = [('apples', 2.0), ('pears', 3.0), ('limes', 4.0)]  ，在已知单价的情况下输出总花费为12.25元，结果正确。

**问题3:shopSmartfunction**

实验验证方法：python shopSmart.py

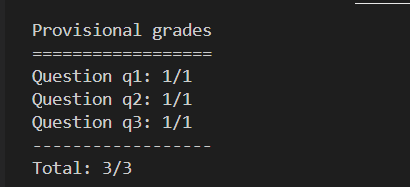
实验结果：



结果评价：当购买的每种水果的重量分别为：

orders = [('apples', 1.0), ('oranges', 3.0)]  ;orders = [('apples', 3.0)] 时，总价较低的商店分别是shop1和shop2，结果正确。

**自动评分：满分**



4、实验结论

1. 实验过程中，对python基础知识有了的了解，并且通过实验加深巩固了所学知识，对利用python实现相应函数功能有了初步的体验；
2. 自动评分显示三个实验结果均正确，获得满分。在实验过程中实现相应功能的同时，保证了实验结果的正确性；

实验1：Search

1、实验目的

1. 完成实验报告中的问题，构建通用的搜索算法并将其应用于Pacman场景，包括到达指定位置以及有效的吃豆；
2. 通过分析吃豆人游戏巩固课堂上所学的搜索算法内容；
3. 复习python语言的使用。

2、实验设计

1. 在本实验中, Pacman 智能体将找到通过迷宫世界的路径, 既包括到达一个指定的位置，也包括高效地搜集食物。我们编辑文件search.py和searchAgents.py，编写一系列吃豆人程序，包括到达指定位置以及有效的吃豆，并将其应用到Pacman场景，完成对相关人工智能功能的完善。
2. 在本实验中，我们对下面8个问题进行研究，针对每个问题提出解决方法，逐步完成吃豆人游戏：

**Q1: Depth First Search**

问题分析：

应用深度优先算法找到一个特定的位置的豆，我们通过depthFirstSearch函数实现深度优先搜索的功能。

解决方案：

深度优先搜索采用堆栈寻找路径，首先从起始结点出发，判断是否为目标结点，若否，寻找与该结点的邻接点，先搜索一条分支上的所有节点，然后再去搜索起始节点的其它分支结点，找出并存进待扩展结点表，等待扩展，每次先判断待扩展结点表是否为空，若否，则从待扩展结点表中取出一个结点进行扩展，并将扩展后的结点存进该表，若是，则返回失败。对于已经搜索过的状态Pacman棋盘上将显示一个叠加物(overlay),并显示出访问的顺序(红色由深到浅). Pacman 在到达目的地的过程中,并不是遍访每个正方形，而是把一种走法显示出来。

程序实现：

1. def depthFirstSearch(problem):
2. "\*\*\* YOUR CODE HERE \*\*\*"
3. visited\_node=[]
4. stack\_node=util.Stack()
5. result=[]
6. if problem.isGoalState(problem.getStartState()):
7. return []
8. stack\_node.push((problem.getStartState(),[]))
9. visited\_node.append(problem.getStartState())
10. while not stack\_node.isEmpty():
11. node=stack\_node.pop()
12. visited\_node.append(node[0])
13. result = node[1]
14. if (problem.isGoalState(node[0])):
15. return result
16. for record in problem.getSuccessors(node[0]):
17. child=record[0]
18. action=record[1]
19. sub\_result=list(result)
20. if not child in visited\_node:
21. sub\_result.append(action)
22. stack\_node.push((child,sub\_result))
23. return result

**Q2: Breadth First Search**

问题分析：

应用宽度优先算法找到一个特定的位置的豆，我们通过breadthFirstSearch函数实现深度优先搜索的功能。

解决方案：

将深度优先算法作为通用搜索算法，不同的搜索方法的不同之处仅仅在于open表的排序不同，因此，宽度优先搜索算法用相应的数据结构对open表进行排序。属于广度优先搜索是一种盲目搜寻法，目的是系统地展开并检查图中的所有节点，以找寻结果。换句话说，它并不考虑结果的可能位置，彻底地搜索整张图，直到找到结果为止，且搜索出来的路径为最短路径。

程序实现：

1. def breadthFirstSearch(problem):
2. """Search the shallowest nodes in the search tree first."""
3. "\*\*\* YOUR CODE HERE \*\*\*"
4. visited\_node=[]
5. frontiers=util.Queue()
6. result=[]
7. frontiers\_node=[]
8. if problem.isGoalState(problem.getStartState()):
9. return []
10. frontiers.push((problem.getStartState(),[]))
11. visited\_node.append(problem.getStartState())
12. frontiers\_node.append(problem.getStartState())
13. while not frontiers.isEmpty():
14. current = frontiers.pop()
15. *#print(current)*
16. visited\_node.append(current[0])
17. result=current[1]
18. if (problem.isGoalState(current[0])):
19. return result
20. for record in problem.getSuccessors(current[0]):
21. child = record[0]
22. action = record[1]
23. sub\_result = list(result)
24. if not child in (visited\_node and frontiers\_node):
25. sub\_result.append(action)
26. frontiers.push((child, sub\_result))
27. frontiers\_node.append(child)
28. return result

**Q3: Uniform Cost Search**

问题分析：

通过修改代价函数，我们鼓励Pacman发现不同路径。例如，有恶魔的区域，我们增加每步的代价，而在食物丰富的区域减少每步的代价，一个理性的Pacman应该相应地调整它的行为。

解决方案：

扩展的是路径消耗g(n)最小的节点n,用优先队列来实现，对解的路径步数不关心，只关心路径总代价。即使找到目标节点也不会结束，而是再检查新路径是不是要比老路径好，确实好，则丢弃老路径。在uniformCostSearch函数中，我们计算每条路径的总代价，将总代价作为优先级进行搜索，待搜索序列存储于队列中。对于每个节点，使用代价函数getCostOfActions计算其所产生的代价，并依次作为搜索的优先级进行搜索。同样的，对于每个节点添加是否被访问的标记。

程序实现：

1. def uniformCostSearch(problem):
2. """Search the node of least total cost first."""
3. "\*\*\* YOUR CODE HERE \*\*\*"
4. visited\_node=[]
5. frontier=util.PriorityQueue()
6. frontiers\_node = []
7. result=[]
8. if problem.isGoalState(problem.getStartState()):
9. return []
10. frontier.push((problem.getStartState(), []),0)
11. *#visited\_node.append(problem.getStartState())*
12. frontiers\_node.append(problem.getStartState())
13. while not frontier.isEmpty():
14. node= frontier.pop()
15. if node[0] in visited\_node:
16. continue
17. visited\_node.append(node[0])
18. *#print("the node is:",node)*
19. result=node[1]
20. if (problem.isGoalState(node[0])):
21. return result
22. for record in problem.getSuccessors(node[0]):
23. child = record[0]
24. action = record[1]
25. sub\_result=list(result)
26. *#if (not child in visited\_node)  :*
27. sub\_result.append(action)
28. frontier.push((child, sub\_result),problem.getCostOfActions(sub\_result))
29. *#print(child,problem.getCostOfActions(sub\_result))*
30. frontiers\_node.append(child)
31. return result

**Q4: A\* Search**

问题分析：

在search.py的aStarSearch函数中实现A\*图搜索。

解决方案：

公式表示为： f(n)=g(n)+h(n),其中 f(n) 是从初始点经由节点n到目标点的估价函数，g(n) 是在状态空间中从初始节点到n节点的实际代价，h(n) 是从n到目标节点最佳路径的估计代价。保证找到最短路径（最优解的）条件，关键在于估价函数f(n)的选取。

程序实现：

1. def aStarSearch(problem, heuristic=nullHeuristic):
2. """Search the node that has the lowest combined cost and heuristic first."""
3. "\*\*\* YOUR CODE HERE \*\*\*"
4. visited\_node=[]
5. frontier=util.PriorityQueue()
6. frontiers\_node = []
7. result=[]
8. if problem.isGoalState(problem.getStartState()):
9. return []
10. frontier.push((problem.getStartState(), []),0)
11. *#visited\_node.append(problem.getStartState())*
12. frontiers\_node.append(problem.getStartState())
13. while not frontier.isEmpty():
14. node= frontier.pop()
15. if node[0] in visited\_node:
16. continue
17. visited\_node.append(node[0])
18. *#print("the node is:",node)*
19. result=node[1]
20. if (problem.isGoalState(node[0])):
21. *#print(result)*
22. *#i = input()*
23. return result
24. for record in problem.getSuccessors(node[0]):
25. child = record[0]
26. action = record[1]
27. sub\_result=list(result)
28. *#if (not child in visited\_node)  :*
29. sub\_result.append(action)
30. frontier.push((child, sub\_result),heuristic(child, problem)+problem.getCostOfActions(sub\_result))
31. *#print(child,problem.getCostOfActions(sub\_result))*
32. frontiers\_node.append(child)
33. print(result)
34. i=input()
35. return result

**Q5: Corners Problem: Representation**

问题分析：

在角落迷宫corner mazes中, 四个角上各有一颗豆。我们新的搜索问题是找到穿过迷宫碰到所有四个角的最短路径(不论在迷宫中是否真有食物). 注意，对于象tinyCorners这样的迷宫, 最短路径不一定总是先找最近的食物! 提示: 通过tinyCorners的最短路径需要28步.

解决方案：

在searchAgents.py中实现CornersProblem搜索问题。需要选择一种状态表示方法，该方法可以对所有必要的信息编码，以便测定所有四个角的点是否达到。

程序实现：

1. def getStartState(self):
2. """
3. Returns the start state (in your state space, not the full Pacman state
4. space)
5. """
6. "\*\*\* YOUR CODE HERE \*\*\*"
7. return (self.startingPosition,[])  *#返回初始位置，以及所经过的角落位置*
8. util.raiseNotDefined()
10. def isGoalState(self, state):
11. """
12. Returns whether this search state is a goal state of the problem.
13. #如果当前位置是角落，并且不在已经经过的表中，将其加入，判断是否是否已经经过四个角落
14. """
15. "\*\*\* YOUR CODE HERE \*\*\*"
16. result=state[1]
17. if state[0] in self.corners:
18. if state[0] not in result:
19. result.append(state[0])
20. if(len(result)==4):
21. return True
22. else:
23. return False
24. util.raiseNotDefined()
26. def getSuccessors(self, state):
27. successors = []
28. for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
29. "\*\*\* YOUR CODE HERE \*\*\*"
30. x, y = state[0]
31. dx, dy = Actions.directionToVector(action)  *#下一步的行动方向*
32. nextx, nexty = int(x + dx), int(y + dy)
33. visited=list(state[1])
34. if not self.walls[nextx][nexty]:  *#如果前方不是墙，则可以移动，否则跳过*
35. nextState = (nextx, nexty)  *#下一个点坐标*
36. cost = 1
37. if nextState in self.corners and nextState not in visited:  *#如果下一个节点是角落但未经过，将他加入，返回后继节点*
38. visited.append(nextState)
39. successors.append(((nextState,visited), action, cost))
41. self.\_expanded += 1 *# DO NOT CHANGE*
42. return successors

**Q6: Corners Problem: Heuristic**

问题分析：

对CornersProblem实现一个启发式搜索cornersHeuristic，完成问题5中的角落搜索问题。

解决方案：

在问题五使用的CornersProblem类中定义cornersHeuristic函数，为角落问题构造启发函数。在cornersHeuristic函数中使用了GetNextNodes函数获取下一个节点，isGoal函数判断是否为目标。

程序实现：

1. def cornersHeuristic(state, problem):
2. """
3. A heuristic for the CornersProblem that you defined.
4. state:   The current search state
5. (a data structure you chose in your search problem)
6. problem: The CornersProblem instance for this layout.
7. This function should always return a number that is a lower bound on the
8. shortest path from the state to a goal of the problem; i.e.  it should be
9. admissible (as well as consistent).
10. """
11. corners = problem.corners *# These are the corner coordinates*
12. walls = problem.walls *# These are the walls of the maze, as a Grid (game.py)*
13. "\*\*\* YOUR CODE HERE \*\*\*"
14. visited=state[1]
15. now\_state=state[0]
16. Heuristic=0
17. last=[]
18. if(problem.isGoalState(state)):
19. return 0
20. for i in corners:
21. if i not in visited:
22. last.append(i)
23. pos=now\_state
24. cost=999999
25. while len(last)!=0:
26. for i in last:
27. if cost>(abs(pos[0] - i[0]) + abs(pos[1] - i[1])):
28. min\_con=i
29. cost=(abs(pos[0] - i[0]) + abs(pos[1] - i[1]))
30. Heuristic+=cost
31. pos=min\_con
32. cost=999999
33. last.remove(min\_con)
34. return Heuristic
35. *#return 0 # Default to trivial solution*

**Q7: Eating All The Dots: Heuristic**

问题分析：

用尽可能少的步数吃掉所有的豆子。完成searchAgents.py文件中的FoodSearchProblem豆子搜索问题。构建合适的启发函数，完成searchAgents.py文件中的foodHeuristic豆子搜索（启发式）问题。

解决方案：

此问题利用之前A\*算法可以很容易找到解。下面在FoodSearchProblem类中定义函数foodHeuristic，构建合适的启发函数完成豆子搜索（启发式）问题。

程序实现：

1. def foodHeuristic(state, problem):
2. position, foodGrid = state
3. "\*\*\* YOUR CODE HERE \*\*\*"
4. lsFoodGrid = foodGrid.asList()
5. last = list(lsFoodGrid)
6. Heuristic = 0
7. cost = 0
8. max\_con=position
9. for i in last:
10. if cost < (abs(position[0] - i[0]) + abs(position[1] - i[1])):
11. max\_con = i
12. cost = (abs(position[0] - i[0]) + abs(position[1] - i[1]))
13. Heuristic = cost
14. diff=position[0]-max\_con[0]
15. count=0
16. for i in last:
17. if diff>0 :
18. if position[0]<i[0]:
19. count+=1
20. if diff<0:
21. if position[0]>i[0]:
22. count+=1
23. if diff==0:
24. if position[0]!=i[0]:
25. count+=1
26. return Heuristic+count

**Q8: Suboptimal Search**

问题分析：

次最优搜索，定义一个优先吃最近的豆子的函数，以此来提高搜索速度。

解决方案：

补充AnyFoodSearchProblem目标测试函数，并在ClosestDotSearchAgent当中添加findPathToClosestDot函数，用于寻找最近的豆子。

程序实现：

indPathToClosestDot函数：

1. def findPathToClosestDot(self, gameState):
2. """
3. Returns a path (a list of actions) to the closest dot, starting from
4. gameState.
5. """
6. *# Here are some useful elements of the startState*
7. startPosition = gameState.getPacmanPosition()
8. food = gameState.getFood()
9. walls = gameState.getWalls()
10. problem = AnyFoodSearchProblem(gameState)
11. "\*\*\* YOUR CODE HERE \*\*\*"
12. last=list(food.asList())
13. closet = last.pop()
14. for i in last:
15. if util.manhattanDistance(startPosition, i) <= util.manhattanDistance(startPosition, closet):
16. closet = i
17. problem.goal = closet
18. actions = search.aStarSearch(problem, manhattanHeuristic)
19. return actions

目标测试函数isGoalState实现：

1. def isGoalState(self, state):
2. """
3. The state is Pacman's position. Fill this in with a goal test that will
4. complete the problem definition.
5. """
6. x,y = state
7. "\*\*\* YOUR CODE HERE \*\*\*"
8. return self.goal == state

3、结果分析

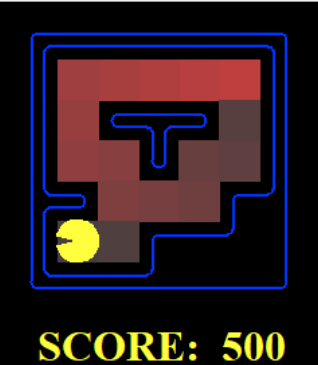
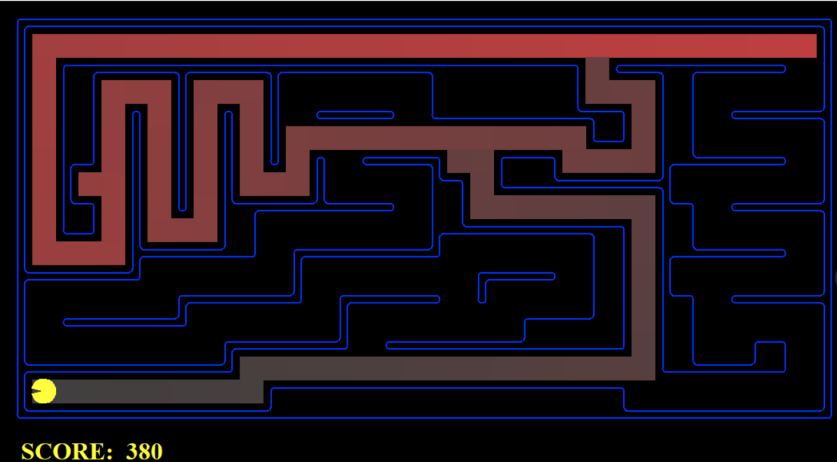
**Q1: Depth First Search**

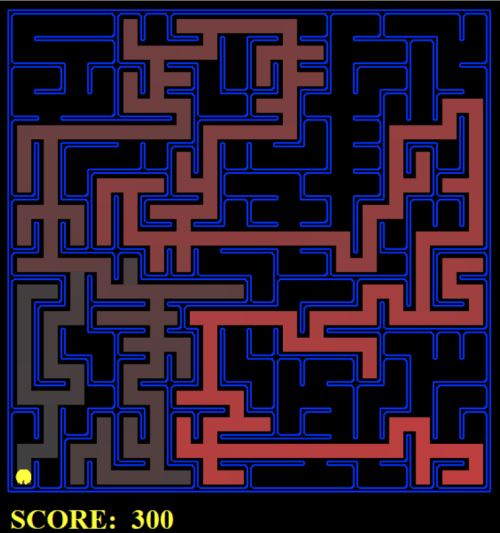
实验验证方法：python pacman.py -l tinyMaze -p SearchAgent

python pacman.py -l mediumMaze -p SearchAgent

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

实验结果：



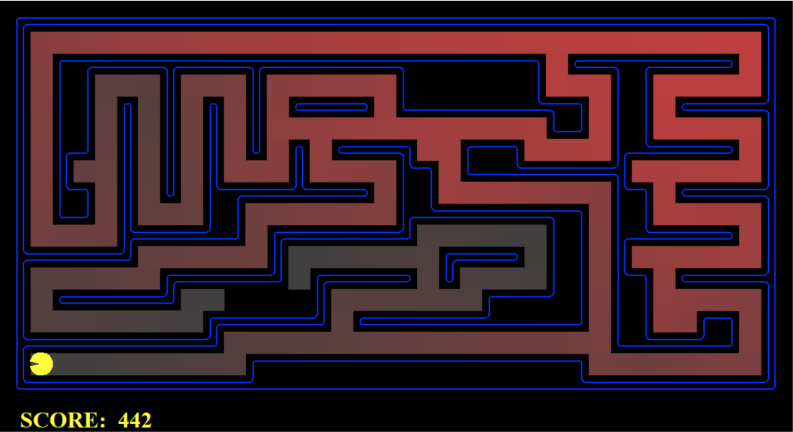
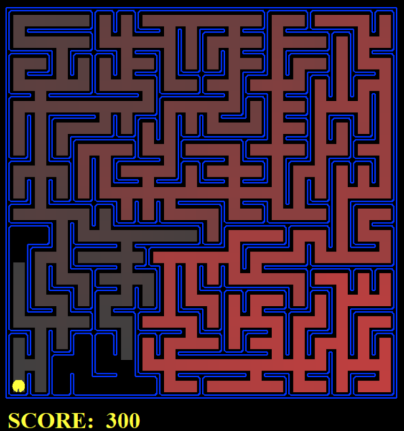
实验结果分析：在0.0秒内找到总开销为分别为10,130，210的路径；搜索节点扩展分别为：15，146，390；得分分别为500,380,和300，pacman都取得了胜利。

**Q2: Breadth First Search**

实验验证方法：python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

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

实验结果：

实验结果分析：在小，中，大三个迷宫中，pacman在0.0秒内找到总开销为分别为68，210的路径；搜索节点扩展分别为：269,620；得分分别为442和300，pacman都取得了胜利。相比深度优先搜索，广度优先搜索在大迷宫中的总开销和得分与中迷宫的相同，但是搜索扩展节点却比深度优先搜索大得多。但是广度优先搜索在中迷宫中表现出优势。

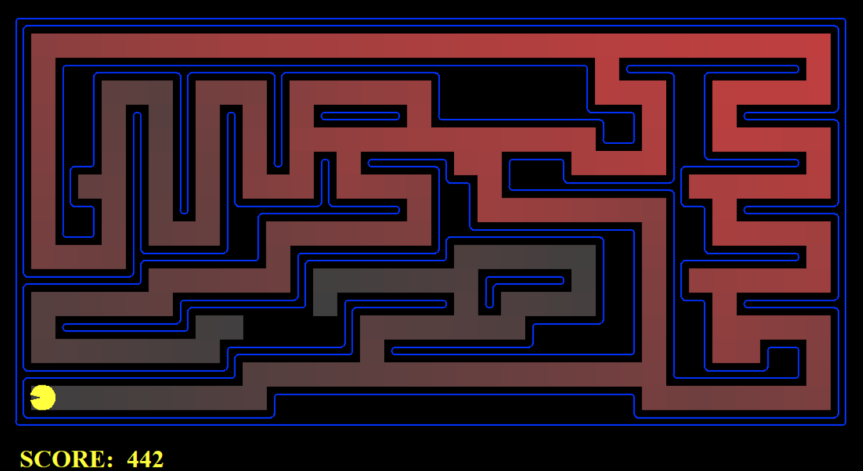
**Q3: Uniform Cost Search**

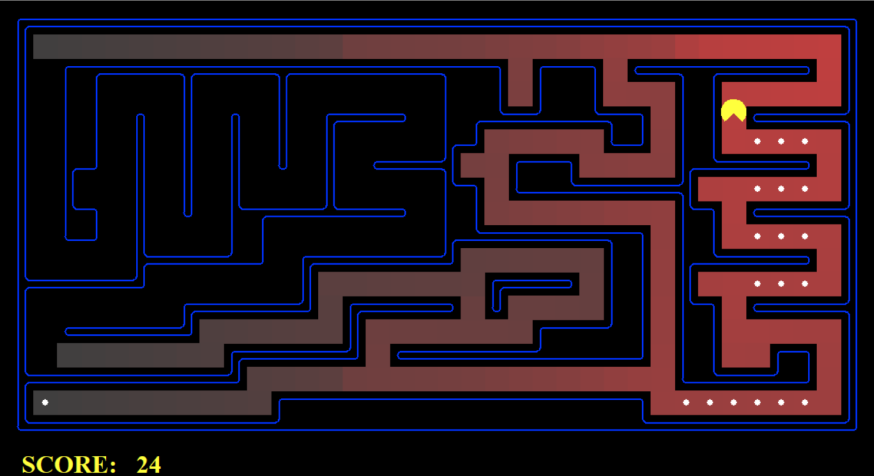
实验验证方法：python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

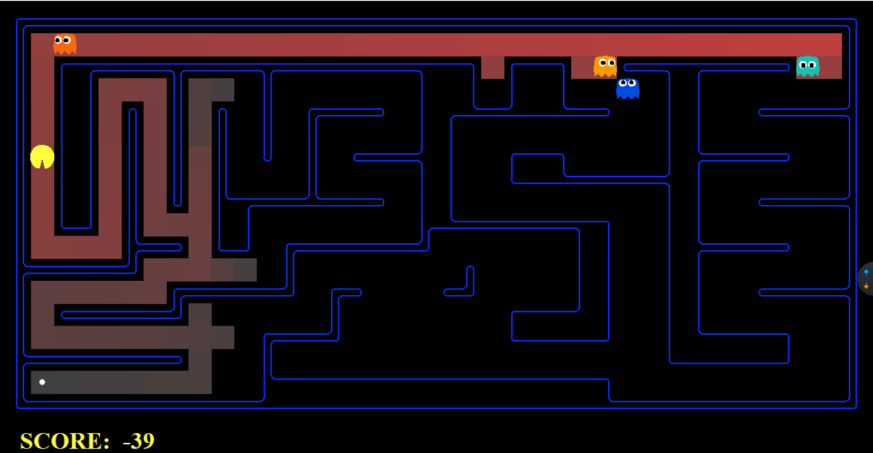
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

实验结果：







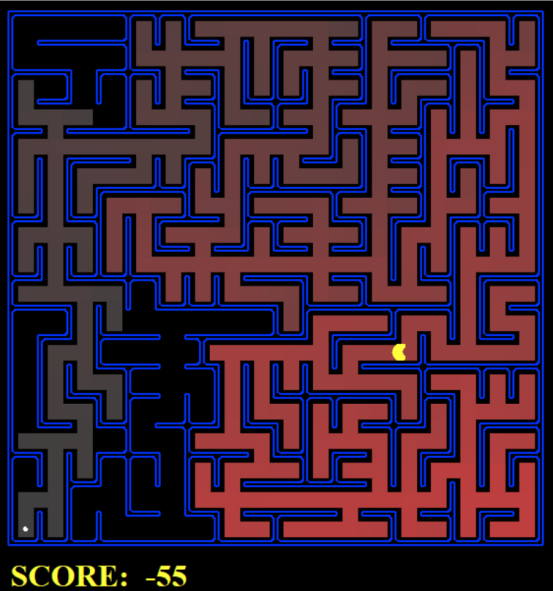
实验结果分析：pacman通过一直代价搜索，在0.0秒内找到总开销为分别为68，1，68719479864的路径；搜索节点扩展分别为：269,186，108；得分分别为442，646和418，pacman都取得了胜利。

**Q4: A\* Search**

实验验证方法：

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

实验结果：



实验结果分析：pacman通过A\*搜索算法在0.1秒内找到总开销为210的路径；搜索节点扩展为：549；得分为300，pacman都得了胜利。

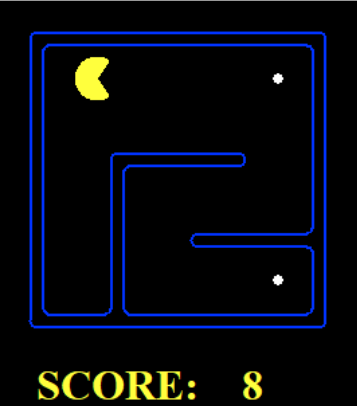
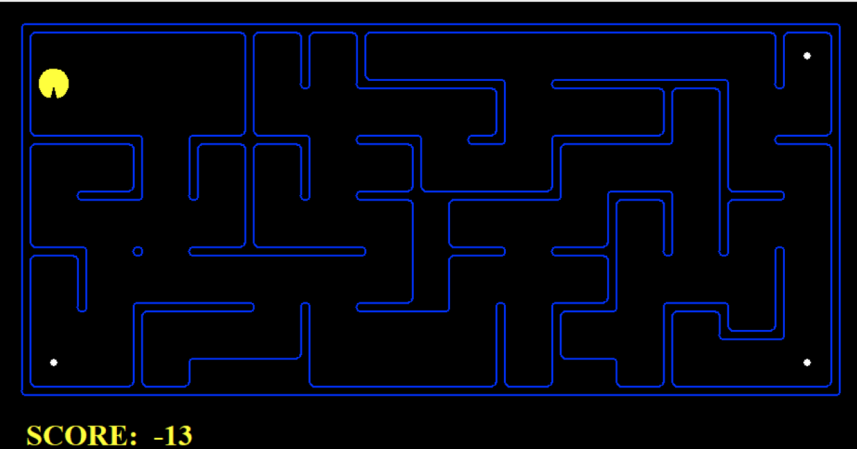
**Q5: Corners Problem: Representation**

实验验证方法：

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

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

实验结果：

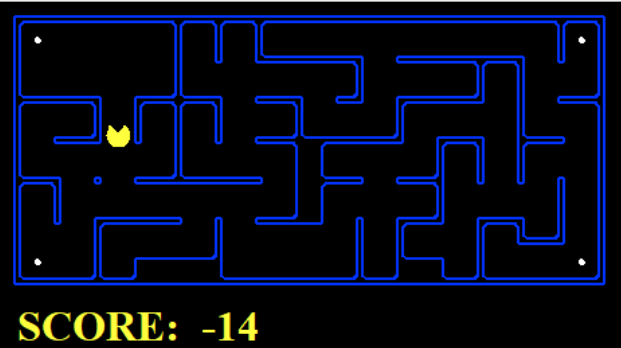
实验结果分析：在小，中迷宫中，pacman通过广度优先搜索算法在0.0到0.3秒内找到总开销分别为28，106的路径；搜索节点扩展分别为：435,2448；得分分别为：512,434，pacman都取得了胜利，实现了吃掉四个角落的豆子的功能。

**Q6: Corners Problem: Heuristic**

实验验证方法：

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

实验结果：



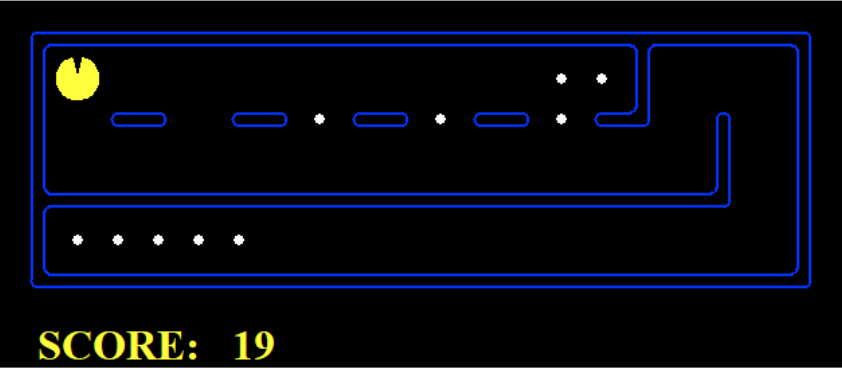
实验结果分析：在中迷宫中，pacman通过A\*搜索算法在0.2秒内找到总开销为106的路径；搜索节点扩展为：901；得分为：434，pacman都取得了胜利，实现了吃掉四个角落的豆子的功能。

**Q7: Eating All The Dots: Heuristic**

实验验证方法：python pacman.py -l testSearch -p AStarFoodSearchAgent

python pacman.py -l trickySearch -p AStarFoodSearchAgent

实验结果：

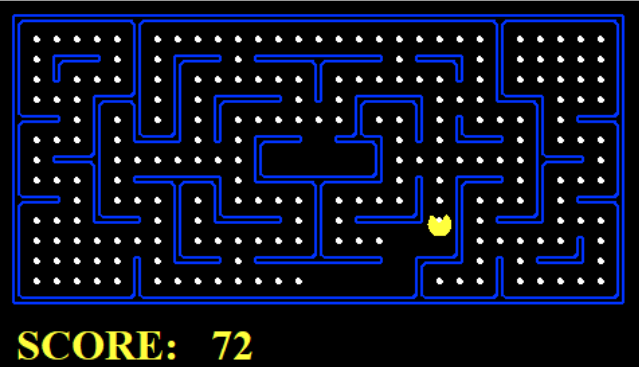
实验结果分析：在小，中迷宫中，pacman通过A\*搜索算法在找到总开销分别为7，60的路径；搜索节点扩展分别为,12，8957；得分分别为：513，570，pacman都取得了胜利，实现了吃掉所有豆子的功能，时间花费分别为0.0秒和8.1秒。

**Q8: Suboptimal Search**

实验验证方法：

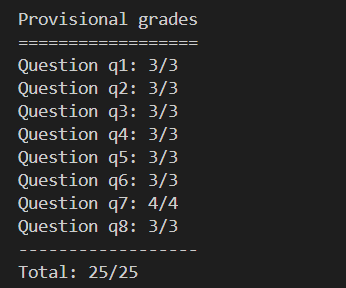
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

实验结果：



实验结果分析：在中迷宫中，pacman通过深度优先搜索算法实现吃掉所有的豆子，搜索节点扩展为346；得分为2364，pacman都取得了胜利。

**自动评分：满分**



4、实验结论

1. 在这个实验中，我们对深度优先搜索、广度优先搜索、代价一致搜索和A\*算法四种搜索方法进行了python代码实现，对这四种方法有了进一步的了解。与深度优先搜索和广度优先上搜索方法相比，代价一致搜索方法的效率更高，在代价一致上搜索方法失效时，等同于广度优先搜索。A\*算法是最有效的直接搜索算法，使用公式进行预处理，这样能够省略大量不必要的搜索路径，提高了效率。
2. 在启发式搜索中，估价函数非常重要，准确的估价函数能够增大搜索函数的效率并且具有相当高的准确性。由此，在启发式搜索中对位置的估价是十分重要的。采用了不同的估价可以有不同的效果。
3. 在本次实验中使用了大量python语言中的类，对之前的python知识有了很大的扩展。

竞赛1：Multi-Agent Pacman

1、实验目的

1. 用自己独特的想法将应用Project 1中实现的搜索算法和问题来处理更困难的场景，包括控制多个Pacman Agent和在时间限制下进行规划；
2. 通过分析吃豆人游戏巩固课堂上所学的搜索算法内容；
3. 复习python语言的使用。

2、实验设计

问题分析：

基本代码与Project 1几乎相同，但做了一些较小的修改，包括支持多个Pacman Agent。竞赛代码以zip文件(minicontest1.zip)的形式提供。一些关键的不同点:

将同时控制环境中的N个吃豆人。你的代码必须能够支持多个Agent。

将考虑一个代价：与吃豆人“思考”的时间有关(计算成本)。更多细节见Scoring部分。

解决方案：

在searchAgents.py中实现CornersProblem搜索问题。需要选择一种状态表示方法，该方法可以对所有必要的信息编码，以便测定所有四个角的点是否达到。

程序实现：

1. **def** isGoalState(self, state):
2. x,y = state
3. "\*\*\* YOUR CODE HERE \*\*\*"
4. **global** tarFood
5. tarFood=[]
6. **global** end
7. **global** foodNum
8. foodNum=[]
9. x,y=state
10. **if** state **not** **in** tarFood **and** self.food[x][y]==True:
11. tarFood.append((x,y))
12. **if** len(tarFood)==foodNum:
13. end=True
14. **return** True
15. **else**:
16. **return** False
17. #util.raiseNotDefined()

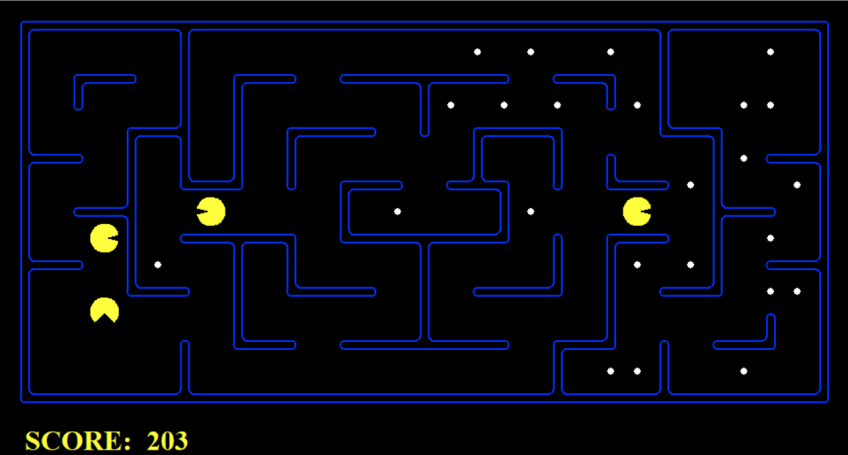
3、结果分析

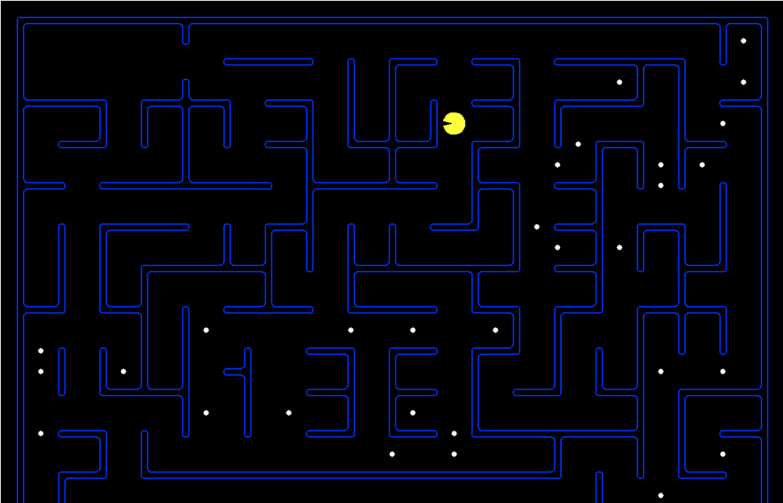
实验验证方法：python pacman.py --pacman myAgents.py

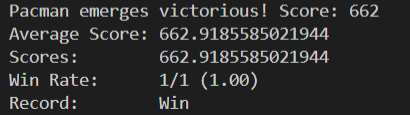
python pacman.py --layout test1.lay

python autograder.py --pacman myAgents.py

实验结果：









实验结果分析：对 myAgents进行了71次竞赛测试，最终比赛结果总成绩为533.470600810522，竞赛获胜。

4、实验结论

1. 本实验对目标测试程序进行了实现，我们同时控制环境中的N个吃豆人，考虑计算成本，令多个吃豆人之间进行竞赛，并实验了71次，我们的吃豆人最终获得胜利。
2. 多个吃豆人之间存在很大的竞争关系，对于每一次竞赛，可能获胜也可能失败，因此进行多次竞赛来测试实验程序的可行性，测试结果可知获胜几率大。
3. 在本次实验中使用了大量python语言中的类，对之前的python知识有了很大的扩展。

实验2：Multi-Agent Search

1、实验目的

* 1. 理解课堂上学习的多智能体对抗搜索算法。
  2. 为经典版本的Pacman游戏设计Agents，也包括ghosts。
  3. 实现minimax、alpha-beta剪枝和expectimax搜索，并进行评估函数设计。

2、实验设计

* 1. 在这个实验中将为经典版本的Pacman游戏设计Agents，也包括ghosts。在此过程中，我们将逐步实现MiniMax、Alpha-Beta Pruning和ExpectiMax搜索，并进行评估函数设计。
  2. 在本实验中，我们对下面5个问题进行研究，针对每个问题提出解决方法，逐步完成吃豆人游戏：

**Q1: Reflex Agent**

问题分析：

设计一个更好的评价函数(evaluationFunction)。评估函数接受当前的状态(currentGameState)和一个合法的Pacman行为(action)并对这一action导致的后继状态返回一个评估值，评估值越大越好。

解决方案：

food.asList()方法返回当前所有未获取食物的列表，当下一步动作不会遇到Ghost时，距离Pacman最近的食物的manhattan距离的相反数作为当前动作导致状态的评估函数值。当下一步会遇到Ghost时，评估函数返回负无穷(float("-inf"))，并且不选择动作(action == ‘Stop’)在我的设计中不被视为可选的动作，因为仅有一个Ghost且没有死角的情况下，停止以躲避Ghost的策略是没有必要的。评估函数的策略如下所示：  
 

（式中MHD(a,b)为a,b之间的曼哈顿距离，foodList为未获取食物列表）

程序实现：

1. def evaluationFunction(self, currentGameState, action):
2. *# Useful information you can extract from a GameState (pacman.py)*
3. successorGameState = currentGameState.generatePacmanSuccessor(action)
4. newPos = successorGameState.getPacmanPosition()
5. newFood = successorGameState.getFood()
6. newGhostStates = successorGameState.getGhostStates()
7. newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
8. "\*\*\* YOUR CODE HERE \*\*\*"
9. food = currentGameState.getFood()
10. sucPos = list(successorGameState.getPacmanPosition())
11. score = float("-inf")
12. foodList = food.asList()
13. if action == 'Stop':
14. return score
15. for ghostState in newGhostStates:
16. if ghostState.getPosition() == tuple(sucPos) and (ghostState.scaredTimer == 0):
17. return score
18. for food in foodList:
19. tmpDistance = -manhattanDistance(sucPos, food)
20. if (tmpDistance > score):
21. score = tmpDistance
22. return score

**Q2: Minimax**

问题分析：

在multiAgents.py中提供的MinimaxAgent类中实现一个基于极大极小值算法的对抗搜索Agent。Miniman算法必须比之前在课堂上看到的算法更通用，可以处理任意数量的Ghost。

解决方案：

Minimax算法 又名极小化极大算法，是一种找出失败的最大可能性中的最小值的算法（即最小化对手的最大得益）。通常以递归形式来实现。

Minimax的基本思想：

1. 当轮到MIN走步时，MAX应该考虑最坏的情况（即f(p)取极小值）
2. 当轮到MAX走步时，MAX应该考虑最好的情况（即f(p)取极大值）
3. 相应于两位棋手的对抗策略，交替使用（1）和（2）两种方法倒推值

在agent列表中Pacman一直是第0位，其他Ghost为1,2...。使用agent参数跟踪不同agent策略（Pacman和Ghost）。使用minValue和maxValue函数跟踪结点alpha-beta值并进行决策。

程序实现：

1. def minValue(self, gameState, agent, depth):
2. current\_best\_value = float("inf")
3. for action in gameState.getLegalActions(agent):
4. suc = gameState.generateSuccessor(agent, action)
5. value = self.minimax(suc, agent + 1, depth)
6. current\_best\_value = min(current\_best\_value, value)
7. return current\_best\_value
8. def maxValue(self, gameState, agent, depth):
9. current\_best\_value = float("-inf")
10. for action in gameState.getLegalActions(agent):
11. suc = gameState.generateSuccessor(agent, action)
12. value = self.minimax(suc, agent + 1, depth)
13. current\_best\_value = max(current\_best\_value, value)
14. if depth == 1 and current\_best\_value == value:
15. self.action = action
16. return current\_best\_value
17. def minimax(self, gameState, agent=0, depth=0):
18. agent = agent % gameState.getNumAgents()
19. if gameState.isWin() or gameState.isLose():
20. return self.evaluationFunction(gameState)
21. if agent == 0:
22. if depth < self.depth:
23. return self.maxValue(gameState, agent, depth+1)
24. else:
25. return self.evaluationFunction(gameState)
26. else:
27. return self.minValue(gameState, agent, depth)
28. def getAction(self, gameState):
29. self.minimax(gameState)
30. return self.action

**Q3: Alpha-Beta Pruning**

问题分析：

在Minimax算法的基础上使用alpha-beta剪枝来更有效地探索极大极小树。

解决方案：

Alpha-Beta剪枝用于裁剪搜索树中没有意义的不需要搜索的树枝，以提高运算速度。表示到目前为止路径上发现的MAX的最佳(即极大值)选择，表示到目前为止路径上发现的MIN的最佳(即极小值)选择。使用minValue和maxValue函数分别追踪最大最小搜索树结点的alpha-beta值，并在minimax函数中不断递归更新和的值。当某个节点的值分别比目前的MAX的或者MIN的的值更差时剪裁此节点剩下的分支(即终止递归调用)。

程序实现：

1. def minValue(self, gameState, depth, agent, a, b):
2. minimum = ["", float("inf")]
3. for action in gameState.getLegalActions(agent):
4. suc = gameState.generateSuccessor(agent, action)
5. value = self.miniMax(suc, depth, agent + 1, a, b)
6. if type(value) is not list:
7. newVal = value
8. else:
9. newVal = value[1]
10. if newVal < minimum[1]:
11. minimum = [action, newVal]
12. if newVal < a:
13. return [action, newVal]
14. b = min(b, newVal)
15. return minimum
16. def maxValue(self, gameState, depth, agent, a, b):
17. maximum = ["", -float("inf")]
18. for action in gameState.getLegalActions(agent):
19. suc = gameState.generateSuccessor(agent, action)
20. value = self.miniMax(suc, depth, agent + 1, a, b)
21. if type(value) is not list:
22. newVal = value
23. else:
24. newVal = value[1]
25. *# real logic*
26. if newVal > maximum[1]:
27. maximum = [action, newVal]
28. if newVal > b:
29. return [action, newVal]
30. a = max(a, newVal)
31. return maximum
32. def miniMax(self, gameState, depth, agent, a, b):
33. if agent >= gameState.getNumAgents():
34. depth += 1
35. agent = 0
36. if (depth == self.depth or gameState.isWin() or gameState.isLose()):
37. return self.evaluationFunction(gameState)
38. elif (agent == 0):
39. return self.maxValue(gameState, depth, agent, a, b)
40. else:
41. return self.minValue(gameState, depth, agent, a, b)
42. def getAction(self, gameState):
43. """
44. Returns the minimax action using self.depth and self.evaluationFunction
45. """
46. actionsList = self.miniMax(gameState, 0, 0, -float("inf"), float("inf"))
47. return actionsList[0]

**Q4: Expectimax**

问题分析：

在multiAgents.py中提供的ExpectimaxAgent类中实现一个基于期望极大算法的对抗搜索Agent。

解决方案：

Minimax和alpha-beta都假设对手会做出最优决策，但实际情况可能并非总是如此。Expectimax算法对于有可能做出次优选择的Agent的概率行为规划非常有用。在Minimax的基础上对Ghost行为决策中引入概率性，当agentIndex>0（即Ghost）时使用expectFinder函数对Agent所有可能行为进行概率评估以改善当对手存在次优选择时Pacman的表现。

程序实现：

1. def expectFinder(self, gameState, depth, agentcounter):
2. expectimax = ["", 0]
3. ghostActions = gameState.getLegalActions(agentcounter)
4. probability = 1.0 / len(ghostActions)
5. if not ghostActions:
6. return self.evaluationFunction(gameState)
7. for action in ghostActions:
8. suc = gameState.generateSuccessor(agentcounter, action)
9. value = self.expectimant(suc, depth, agentcounter + 1)
10. if type(value) is list:
11. newVal = value[1]
12. else:
13. newVal = value
14. expectimax[0] = action
15. expectimax[1] += newVal \* probability
16. return expectimax
17. def maxValue(self, gameState, depth, agentcounter):
18. maximum = ["", -float("inf")]
19. actions = gameState.getLegalActions(agentcounter)
20. if not actions:
21. return self.evaluationFunction(gameState)
22. for action in actions:
23. suc = gameState.generateSuccessor(agentcounter, action)
24. value = self.expectimant(suc, depth, agentcounter + 1)
25. if type(value) is not list:
26. newVal = value
27. else:
28. newVal = value[1]
29. if newVal > maximum[1]:
30. maximum = [action, newVal]
31. return maximum
32. def expectimant(self, gameState, depth, agentcounter):
33. if agentcounter >= gameState.getNumAgents():
34. depth += 1
35. agentcounter = 0
36. if depth == self.depth or gameState.isWin() or gameState.isLose():
37. return self.evaluationFunction(gameState)
38. elif (agentcounter == 0):
39. return self.maxValue(gameState, depth, agentcounter)
40. else:
41. return self.expectFinder(gameState, depth, agentcounter)
42. def getAction(self, gameState):
43. """
44. Returns the expectimax action using self.depth and self.evaluationFunction
45. All ghosts should be modeled as choosing uniformly at random from their
46. legal moves.
47. """
48. "\*\*\* YOUR CODE HERE \*\*\*"
49. actionsList = self.expectimant(gameState, 0, 0)
50. return actionsList[0]

**Q5: Evaluation Function**

问题分析：

在函数betterEvaluationFunction中为Pacman编写一个更好的评估函数。评估函数应该评估states，而不是像Q1中评估函数那样评估action。在深度2搜索中，评估函数应该在一半以上的时间里清除带有随机幽灵的小型经典布局，并仍然以合理的速度运行(为了获得全部积分，吃豆人在获胜时应该平均获得1000分左右)。

解决方案：

与Q1类似，currentGameState.getFood().asList()方法返回当前所有未获取食物的列表，当前距离Pacman最近的食物的manhattan距离的相反数作为当前状态的评估函数值。评估函数的策略如下所示：  
 

（式中MHD(a,b)为a,b之间的曼哈顿距离，foodList为未获取食物列表）

程序实现：

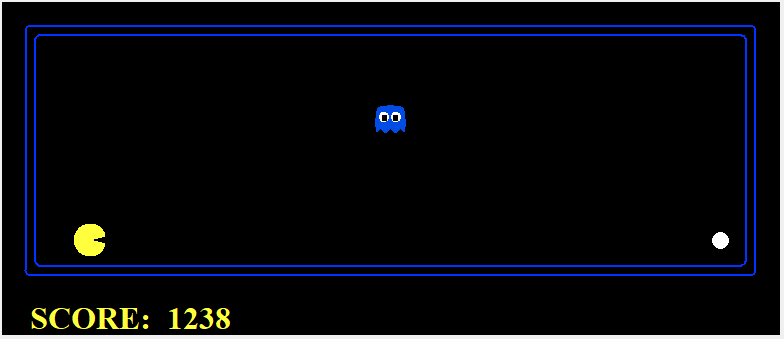
1. def betterEvaluationFunction(currentGameState):
2. """
3. Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
4. evaluation function (question 5).
5. DESCRIPTION: <write something here so we know what you did>
6. """
7. "\*\*\* YOUR CODE HERE \*\*\*"
8. position = list(currentGameState.getPacmanPosition())
9. foodList = currentGameState.getFood().asList()
10. scoreList = []
11. for food in foodList:
12. scoreList.append(-manhattanDistance(position, food))
13. if not scoreList:
14. scoreList.append(0)
15. bestscore = max(scoreList)
16. return currentGameState.getScore() + bestscore

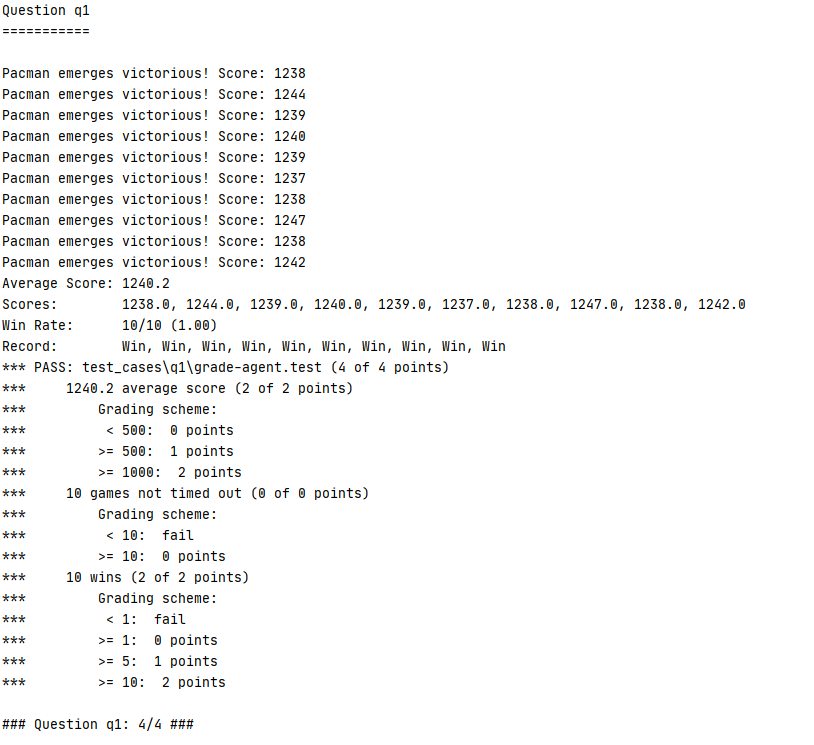
3、结果分析

**Q1: Reflex Agent**

实验验证方法：python autograder.py -q q1

实验结果：



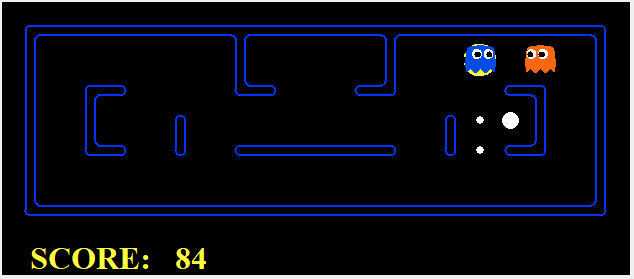


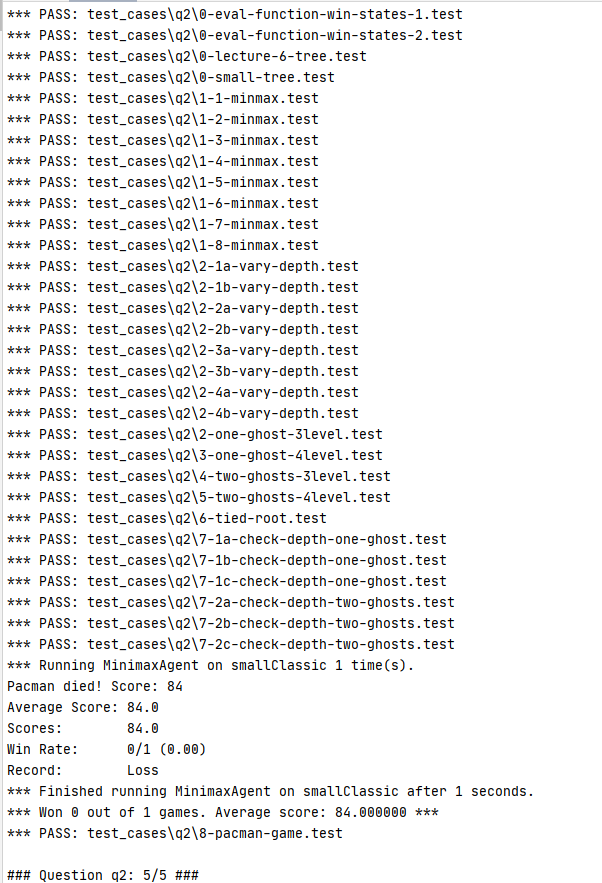
实验结果分析：平均得分1240.2,10次测试中pacman都取得了胜利。

**Q2: Minimax**

实验验证方法：python autograder.py -q q2

实验结果：





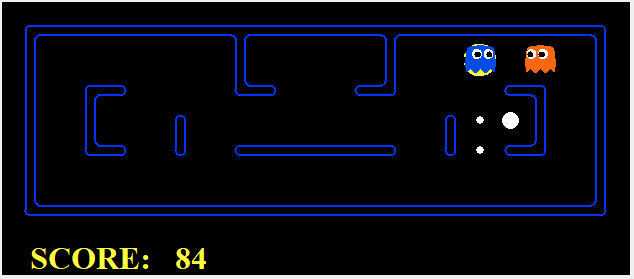
实验结果分析：

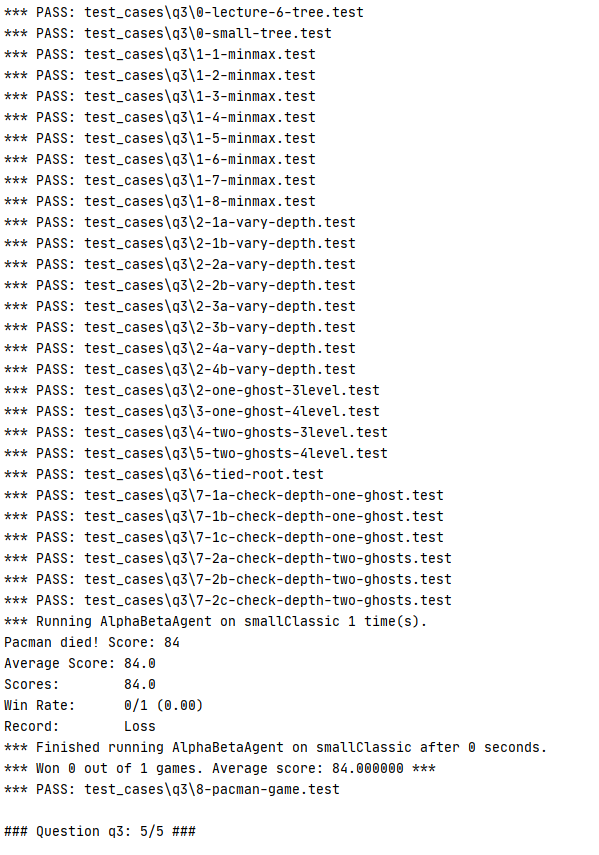
Pacman与Ghost均使用Minimax进行决策，Pacman最终被吃掉，算法通过验证。

**Q3: Alpha-Beta Pruning**

实验验证方法：python autograder.py -q q3

实验结果：





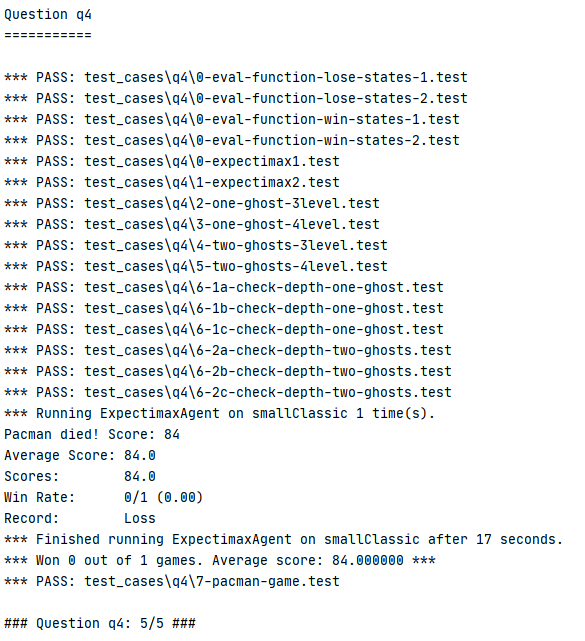
实验结果分析：

Pacman与Ghost均使用Minimax进行决策，Pacman最终被吃掉，算法通过验证。

**Q4: Expectimax**

实验验证方法：python autograder.py -q q4

实验结果：



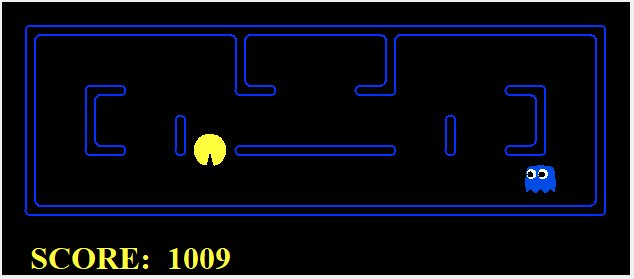
实验结果分析：

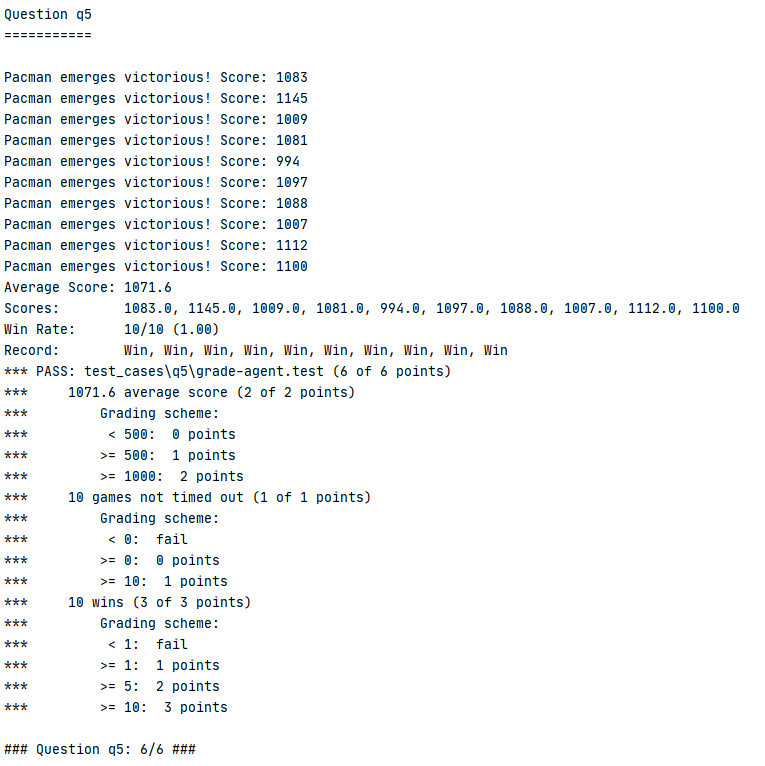
Pacman使用Expectimax进行决策，Pacman最终被吃掉，算法通过验证。

**Q5: Evaluation Function**

实验验证方法：python autograder.py -q q5

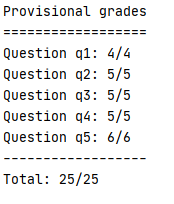
实验结果：





实验结果分析：Pacman赢得了10次比赛，平均分1071.6，并均未超时。

**自动评分：满分**

****

4、实验结论

* 1. 在该实验中我们实现了Minimax算法，并使用alpha-beta剪枝对搜索进行了优化，然后实现了期望极大搜索算法并用于Pacman决策。
  2. 在Pacman决策过程中，优秀的评估函数对Pacman的行为结果有决定性的影响。在实际agent设计过程中，评估函数的设计往往对搜索结果和决策效率起到关键性的作用。

竞赛2：Multi-Agent Adversarial Pacman

1. 实验目的

第一场竞赛和第二场竞赛之间的主要变化是竞赛2是一种对抗式游戏，包括两支队伍相互竞争。队伍将尝试在地图的远端吃食物，同时保护家乡的食物。

我方Agent以Ghost的形式出现在敌方主场，而Pacman则出现在对手的主场。同时，当我方变成Ghost的时候，可以吃掉对手。如果Pacman在到达自己的一侧之前就被Ghost吃掉了，他就会爆炸成一团食物点，然后这些食物点就会被放回棋盘上。

1. 实验设计

在baselineTeam.py中实现了基本的编队Agent，我们需要编写多种攻守策略的Agent，通过creaetTeam函数实现编队。通常的做法是将两只Agent分别赋予保守型(Defensive)和激进型(Offensive)的行为模式，保守型的Ghost针对越界的敌方Pacman进行捕捉，而激进型Pacman尝试入侵敌方区域吃豆并躲避捕捉尝试返回。

我在继承baselineTeam.py中的ReflexCaptureAgent和captureAgents.py中的CaptureAgent基础上进行了一定的改进，并参考了baselineTeam.py中的OffensiveReflexAgent和DefensiveReflexAgent的设计，同时查阅了一定的相关资料，最终设计了我们的编队策略。

在父类DummyAgent中，我们追加了food\_eaten变量追踪Pacman当前持有的食物。当OffensiveAgent成功入侵并获得1个及以上的豆时，控制其躲避地方Ghost并返回。

在评估函数evalutate中，Agent针对当前游戏状态gameState和一个动作返回一个评估值，评估值为features及其权重weights的乘积。features是经过设计的针对对抗竞赛设计的特征，在baselineTeam中预设了successorScore、distanceToFood、numInvaders、onDefense、invaderDistance、stop、reverse等特征，通过对这些特征赋权，以更新OffensiveAgent和DefensiveAgent的行为模式。

在OffensiveAgent的设计中增加了defenders、distanceToHome和invaderDistance三个特征分别用以对敌方Pacman、吃豆后距离己方距离及入侵者距离进行赋权。同我们在实验2的Q1、Q5中的设计，赋权使用曼哈顿距离的相反数以实现当距离越近时权值越大。

在DefensiveAgent的设计中增加了defenders用于针对敌方Pacman进行防守。

程序实现：

1. *##########*
2. *# Agents #*
3. *##########*
4. class DummyAgent(CaptureAgent):
5. def registerInitialState(self, gameState):
6. self.food\_eaten = 0
7. self.start = gameState.getAgentPosition(self.index)
8. CaptureAgent.registerInitialState(self, gameState)
9. def chooseAction(self, gameState):
10. actions = gameState.getLegalActions(self.index)
11. *# You can profile your evaluation time by uncommenting these lines*
12. *# start = time.time()*
13. values = [self.evaluate(gameState, a) for a in actions]
14. *# print 'eval time for agent %d: %.4f' % (self.index, time.time() - start)*
15. maxValue = max(values)
16. bestActions = [a for a, v in zip(actions, values) if v == maxValue]
17. next\_action = random.choice(bestActions)
18. nextGameState = self.getSuccessor(gameState, next\_action)
19. foodLeft = len(self.getFood(gameState).asList())
20. if not nextGameState.getAgentState(self.index).isPacman:
21. self.food\_eaten = 0
22. self.food\_eaten += len(self.getFood(gameState).asList()) - len(self.getFood(nextGameState).asList())
23. if foodLeft <= 2:
24. bestDist = 9999
25. for action in actions:
26. successor = self.getSuccessor(gameState, action)
27. pos2 = successor.getAgentPosition(self.index)
28. dist = self.getMazeDistance(self.start, pos2)
29. if dist < bestDist:
30. bestAction = action
31. bestDist = dist
32. return bestAction
33. return next\_action
34. def getSuccessor(self, gameState, action):
35. """
36. Finds the next successor which is a grid position (location tuple).
37. """
38. successor = gameState.generateSuccessor(self.index, action)
39. pos = successor.getAgentState(self.index).getPosition()
40. if pos != nearestPoint(pos):
41. *# Only half a grid position was covered*
42. return successor.generateSuccessor(self.index, action)
43. else:
44. return successor
45. def evaluate(self, gameState, action):
46. """
47. Computes a linear combination of features and feature weights
48. """
49. features = self.getFeatures(gameState, action)
50. weights = self.getWeights(gameState, action)
51. return features \* weights
52. def getFeatures(self, gameState, action):
53. """
54. Returns a counter of features for the state
55. """
56. features = util.Counter()
57. successor = self.getSuccessor(gameState, action)
58. features['successorScore'] = self.getScore(successor)
59. return features
60. def getWeights(self, gameState, action):
61. """
62. Normally, weights do not depend on the gamestate.  They can be either
63. a counter or a dictionary.
64. """
65. return {'successorScore': 1.0}
66. class OffensiveAgent(DummyAgent):
67. def getFeatures(self, gameState, action):
68. features = util.Counter()
69. successor = self.getSuccessor(gameState, action)
70. foodList = self.getFood(successor).asList()
71. features['successorScore'] = -len(foodList)
72. myPos = successor.getAgentState(self.index).getPosition()
73. enemies = [successor.getAgentState(i) for i in self.getOpponents(successor)]
74. invaders = [a for a in enemies if a.isPacman and a.getPosition() != None]
75. defenders = [a for a in enemies if not a.isPacman and a.getPosition() != None]
76. if len(foodList) > 0 and self.food\_eaten == 0 and len(invaders) == 0:
77. minFoodDistance = min([self.getMazeDistance(myPos, food) for food in foodList])
78. features['distanceToFood'] = minFoodDistance
79. for ghost in defenders:
80. if ghost.scaredTimer == 0:
81. if len(defenders) > 0:
82. features['defenders'] = min(self.getMazeDistance(myPos, ghost.getPosition()) for ghost in defenders)
83. else:
84. features['defenders'] = 0
85. if features['defenders'] > 2:
86. features['defenders'] = 0
87. if self.food\_eaten > 0:
88. minHomeDistance = min([self.getMazeDistance(myPos, self.start)])
89. features['distanceToHome'] = minHomeDistance
90. if len(invaders) > 0:
91. dists = [self.getMazeDistance(myPos, a.getPosition()) for a in invaders]
92. features['invaderDistance'] = min(dists)
93. if action == Directions.STOP: features['stop'] = 1
94. return features
95. def getWeights(self, gameState, action):
96. return {'successorScore': 100, 'distanceToFood': -1, 'defenders': -10, 'distanceToHome': -10,
97. 'invaderDistance': -10, 'stop': -100}
98. class DefensiveAgent(DummyAgent):
99. def getFeatures(self, gameState, action):
100. features = util.Counter()
101. successor = self.getSuccessor(gameState, action)
102. myState = successor.getAgentState(self.index)
103. myPos = myState.getPosition()
104. *# Computes whether we're on defense (1) or offense (0)*
105. features['onDefense'] = 1
106. if myState.isPacman: features['onDefense'] = 0
107. *# Computes distance to invaders we can see*
108. enemies = [successor.getAgentState(i) for i in self.getOpponents(successor)]
109. invaders = [a for a in enemies if a.isPacman and a.getPosition() != None]
110. defenders = [a for a in enemies if not a.isPacman and a.getPosition() != None]
111. features['numInvaders'] = len(invaders)
112. if len(invaders) > 0:
113. dists = [self.getMazeDistance(myPos, a.getPosition()) for a in invaders]
114. features['invaderDistance'] = min(dists)
115. if len(defenders) > 0 and len(invaders) == 0:
116. dists = [self.getMazeDistance(myPos, a.getPosition()) for a in defenders]
117. features['defenders'] = min(dists)
118. if action == Directions.STOP: features['stop'] = 1
119. rev = Directions.REVERSE[gameState.getAgentState(self.index).configuration.direction]
120. if action == rev: features['reverse'] = 1
121. return features
122. def getWeights(self, gameState, action):
123. return {'numInvaders': -1000, 'onDefense': 100, 'invaderDistance': -10, 'defenders': -10, 'stop': -100,
124. 'reverse': -2}

3、结果分析

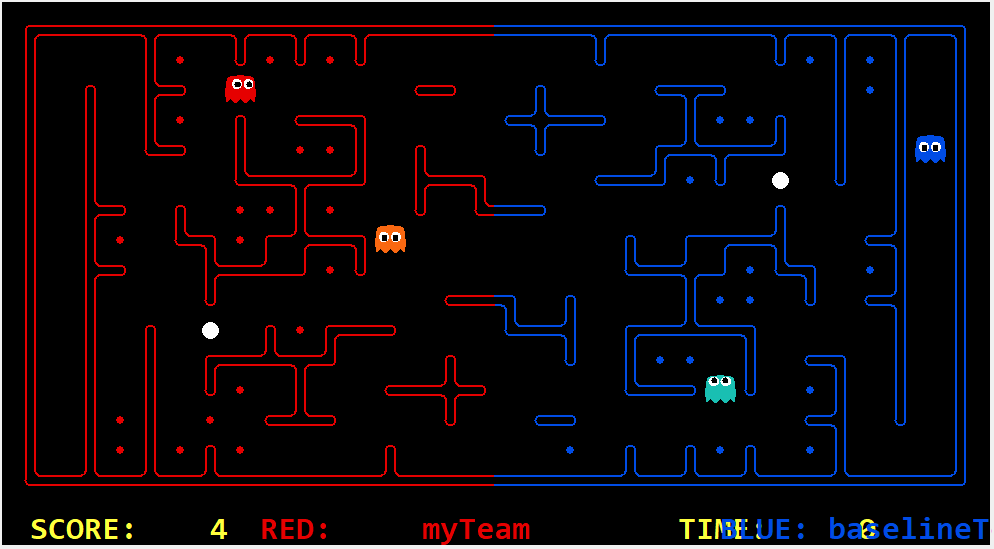
实验验证方法：

python capture.py -r myTeam -b baselineTeam

python capture.py -r myTeam -b baselineTeam -l RANDOM1(2,3,...)

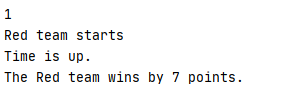
实验结果：

default layout：



2021-01-24_091449

RANDOM1：



RANDOM10：

2021-01-24_091935

RANDOM100：

2021-01-24_092150

RANDOM1000：

2021-01-24_092644

RANDOM10000：

2021-01-24_093003

RANDOM100000：

2021-01-24_093024

RANDOM1000000：

2021-01-24_093039

RANDOM10000000：

2021-01-24_093450

RANDOM100000000：

2021-01-24_093518

RANDOM1000000000：

2021-01-24_093531

实验结果分析：

对 myTeam进行了11次竞赛测试，最终比赛结果总成绩为80，平均胜点7.27，竞赛全部获胜。

4、实验结论

* 1. 本实验myTeam进行了实现，我们同时控制环境中的2个Agent，在11张不同的地图上与baselineTeam进行竞赛，我方的编队最终获得胜利。
  2. 在baselineTeam.py中实现了基本的编队Agent，我们需要编写多种攻守策略的Agent，通过creatTeam函数实现编队，将两只Agent分别赋予保守型(Defensive)和激进型(Offensive)的行为模式，保守型的Ghost针对越界的敌方Pacman进行捕捉，而激进型Pacman尝试入侵敌方区域吃豆并躲避捕捉尝试返回。
  3. 本次实验再一次证明评估函数对启发式Agent设计中的重要性。通过设计恰当的评估函数，Agent能够对环境有更好的评估，从而在竞赛中取得胜利。

团队成员贡献

尤冠杰

* 组长：协调组员分工、进度把控、资料收集和报告撰写。
* 实验2算法设计、竞赛2算法设计。

孙懿豪

* 实验1算法设计、竞赛1算法设计。

姚缤

* 实验2算法设计、竞赛2算法设计。

李强

* 实验1算法设计、竞赛1算法设计。

谭沁

* 实验1算法设计、实验2算法设计。

每个成员都独立完成了实验0，但具体代码和实现逻辑不一一列出，仅列出组长的实现。