人 工 智 能 实 验 报 告

实验名称：　　　　Pacman 实验　　　　

姓　　名：	黄结伦	学　　号：	202002422003
指导教员：	李莎莎	职　　称：	研究员
实 验 室：	102-201	实验日期：	2019.11-2020.1

国防科学技术大学训练部制

Pacman 实验1

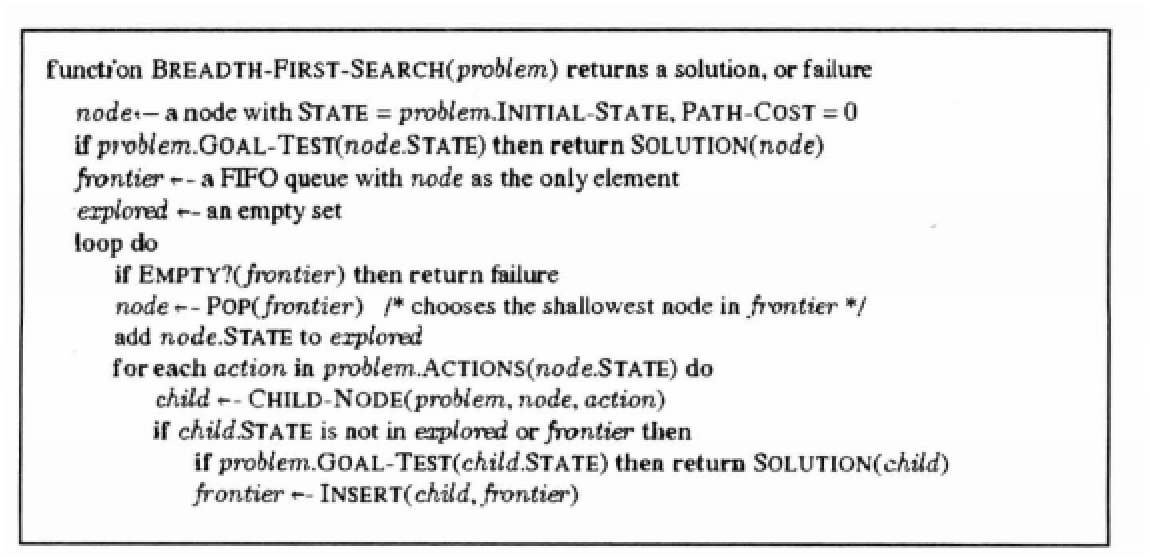
一.实验目的

1.加深对经典的DFS，BFS，一致代价搜索，A*搜索的理解，并掌握其初步的使用方法

二.实验原理

1.宽度优先搜索

宽度优先搜索（bread-first search)是简单搜索策略，先扩展根节点，接着扩展根节点的所有后继，然后再扩展它们的后继，以此类推。其伪代码如下



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

下图显示了一个简单二叉树的搜索过程

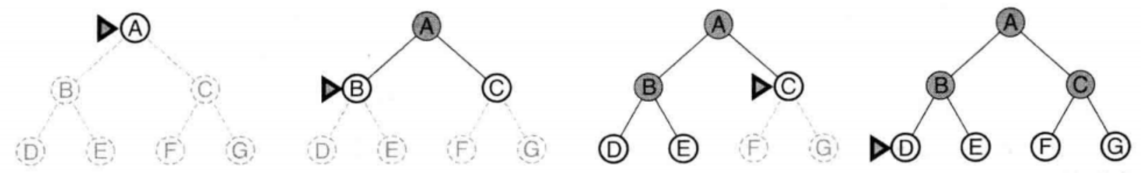
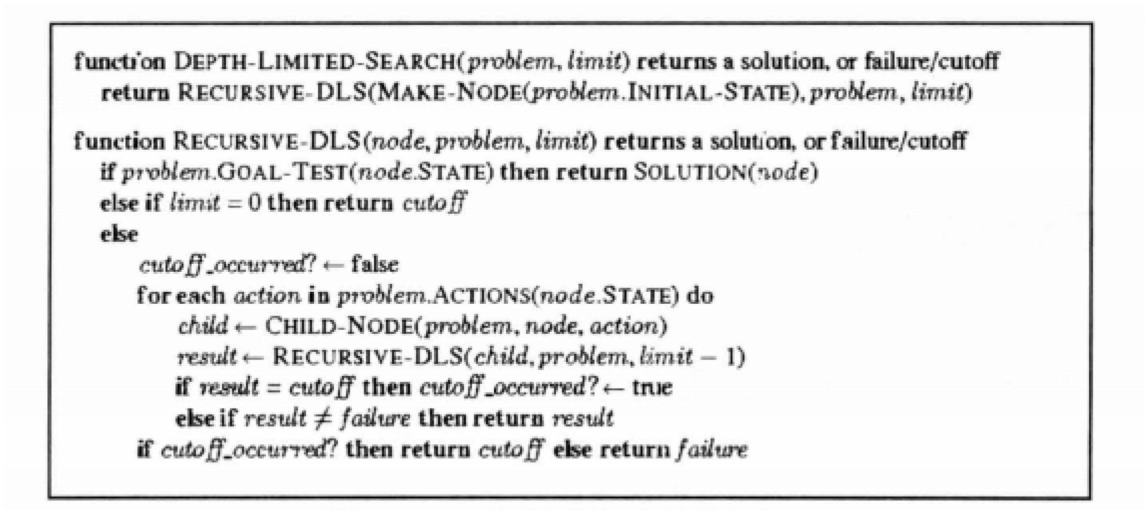


图 3.12　一棵简单二叉树上的宽度优先搜索。在每个阶段，用一个记号指出下一个将要扩展的结点

2.深度优先搜索

深度优先搜索(depth-first search)总是扩展搜索树的当前边缘结点集中最深的节点。搜索很快推进到搜索树的最深层，那里的结点没有后继。当那些结点扩展完之后，就从边缘结点集中去掉，然后搜索算法回溯到下一个还有未扩展后继的深度稍浅的结点。其伪代码如下



```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
     cutoff_occurred? ← false
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit − 1)
        if result = cutoff then cutoff_occurred? ← true
        else if result ≠ failure then return result
     if cutoff_occurred? then return cutoff else return failure
```

下图显示了一个简单的二叉树搜索过程

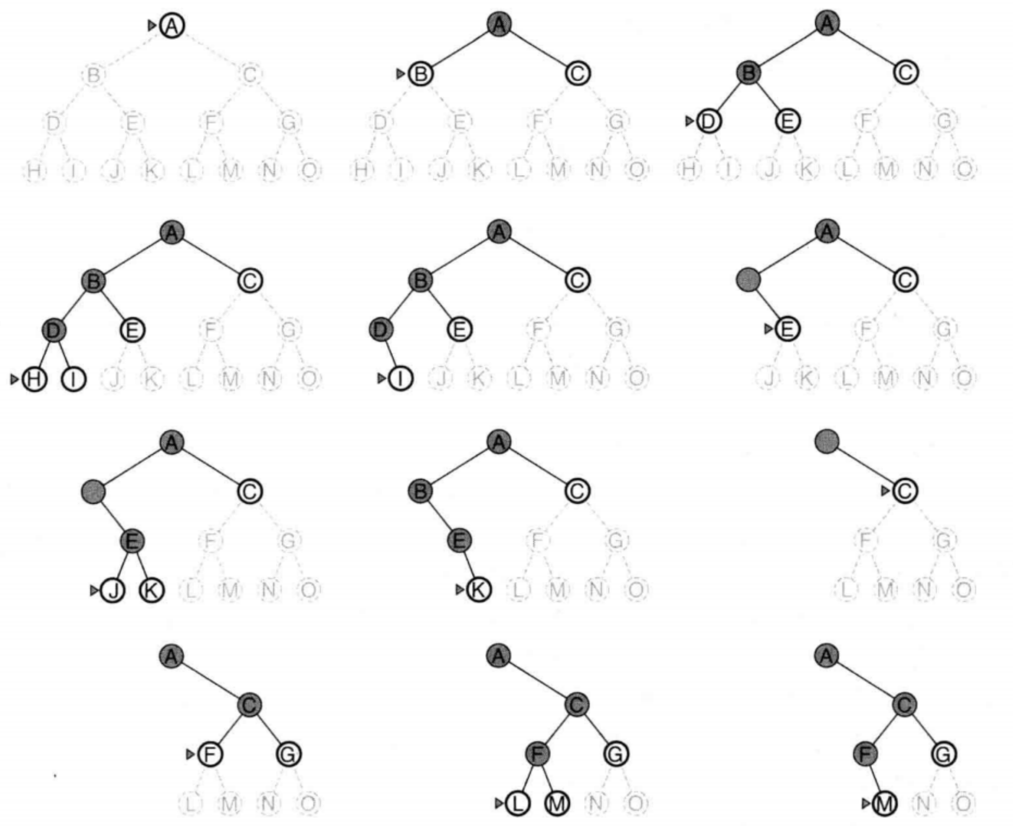
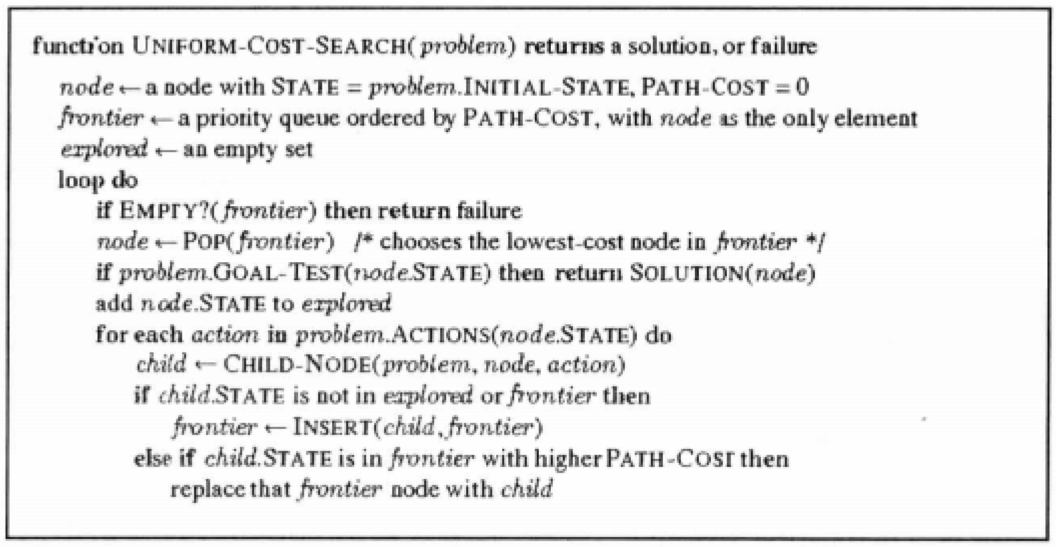


图 3.16　二叉树的深度优先搜索。未探索区域用浅灰色表示。已经被探索并且在边缘中没有后代的结点可以从内存中删除。第三层的结点没有后继并且 M 是唯一的目标结点

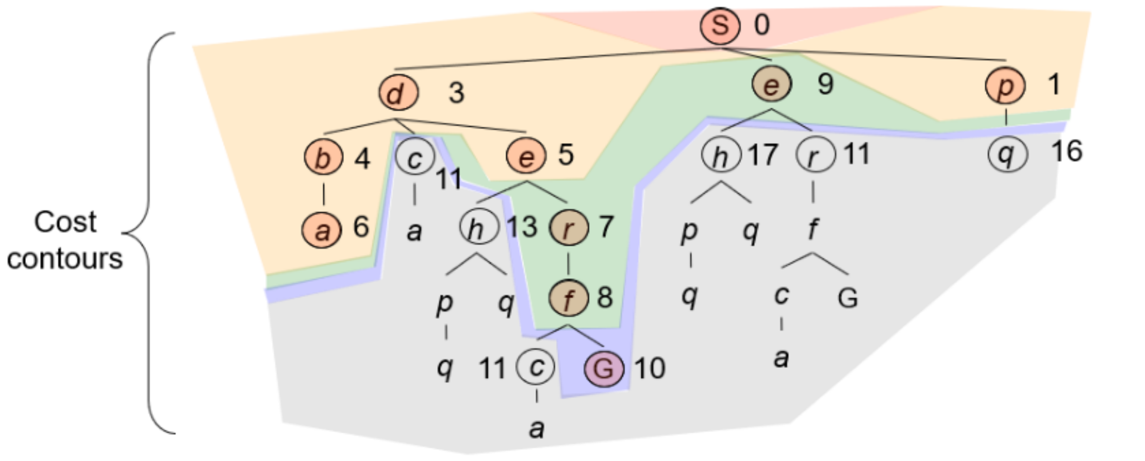
3.一致代价搜索

一致代价搜索总是扩展路径消耗最小的节点N。N点的路径消耗等于前一节点N-1的路径消耗加上N-1到N节点的路径消耗。其伪代码如下



```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

下图举了一个简单的例子说明一致代价搜索



4.A*搜索

A*搜索对结点的评估结合了g(n),即到达此结点已经花费的代价,和A(n),从该结点到目标结点所花代价:

`f(n)=g(n)+h(n)`

由于g(n)是从开始结点到结点n的路径代价,而A(n)是从结点n到目标结点的最小代价路径的估计值,因此

`f(n)= 经过结点n的最小代价解的估计代价`

这样,如果我们想要找到最小代价的解,首先扩展g(n)+ A(n)值最小的结点是合理的.可以发现这个策略不仅仅合理:假设启发式函数H(n)满足特定的条件,*A搜索既是完备的也是最优的.算法与一致代价搜索类似,除了A*使用g+h而不是g.

下图举了一个简单的例子说明A*搜索的过程

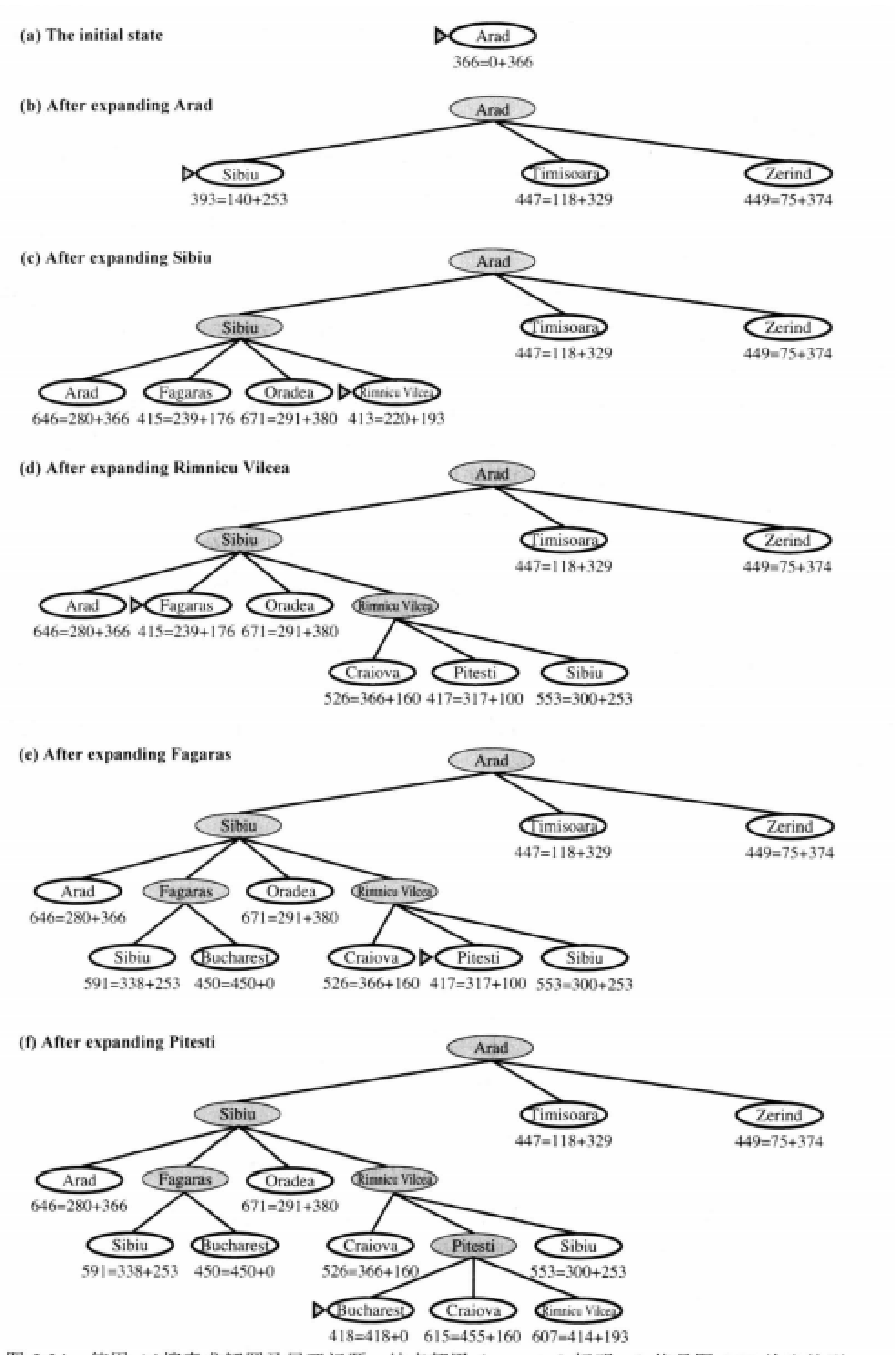


图 3.24 使用 A*搜索求解罗马尼亚问题。结点都用 $f = g + h$ 标明。h 值是图 3.22 给出的到 Bucharest 的直线距离

三.实验内容

1.完成search文件夹中search.py中的depthFirstSearch，breadFirstSearch，uniformCostSearch，aStarSearch四个函数。

在编写完代码后，可在终端中的search目录下执行`python2 .\autograder.py` 命令，即可查看是否通过！

四.实验结果

搜索算法中四个函数的具体实现如下

1.depthFirstSearch

```
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 ***"
    visited_node = []
    myStack= util.Stack()
    actions = []
    s = problem.getStartState()
    if problem.isGoalState(s):
        return actions
    myStack.push((s, actions))
    while not myStack.isEmpty():
        state = myStack.pop()
        if state[0] in visited_node:
            continue
        visited_node.append(state[0])
        actions = state[1]
        if (problem.isGoalState(state[0])):
            return actions
        for successor in problem.getSuccessors(state[0]):
            child_state = successor[0]
            action = successor[1]
            sub_action = list(actions)
            if not child_state in visited_node:
                sub_action.append(action)
                myStack.push((child_state, sub_action))
    return actions
    util.raiseNotDefined()
```

2.breadFirstSearch

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    visited_node = []
    myQueue = util.Queue()
    actions = []
    s = problem.getStartState()
    if problem.isGoalState(s):
        return actions
    myQueue.push((s, actions))
    while not myQueue.isEmpty():
        state = myQueue.pop()
        if state[0] in visited_node:
            continue
        visited_node.append(state[0])
        actions = state[1]
        if (problem.isGoalState(state[0])):
            return actions
        for successor in problem.getSuccessors(state[0]):
            child_state = successor[0]
            action = successor[1]
            sub_action = list(actions)
            if not child_state in visited_node:
                sub_action.append(action)
                myQueue.push((child_state, sub_action))
    return actions
    util.raiseNotDefined()
```

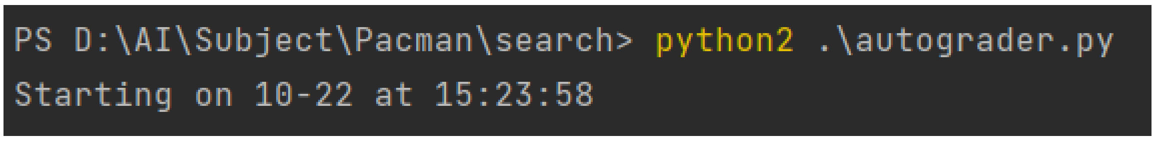
3.uniformCostSearch

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    visited_node = []
    mypriorityQueue = util.PriorityQueue()
    actions = []
    s = problem.getStartState()
    if problem.isGoalState(s):
        return actions
    mypriorityQueue.push((s, actions), 0)
    while not mypriorityQueue.isEmpty():
        state = mypriorityQueue.pop()
        if state[0] in visited_node:
            continue
        visited_node.append(state[0])
        actions = state[1]
        if (problem.isGoalState(state[0])):
            return actions
        for successor in problem.getSuccessors(state[0]):
            child_state = successor[0]
            action = successor[1]
            sub_action = list(actions)
            if not child_state in visited_node:
                sub_action.append(action)
                mypriorityQueue.push((child_state, sub_action),
problem.getCostOfActions(sub_action))
    return actions
    util.raiseNotDefined()
```

4.aStarSearch

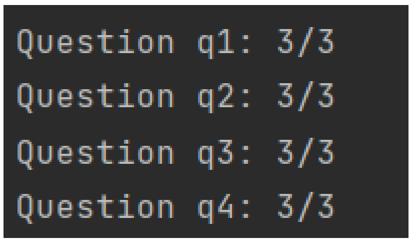
```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    visited_node = []
    mypriorityQueue = util.PriorityQueue()
    actions = []
    s = problem.getStartState()
    if problem.isGoalState(s):
        return actions
    mypriorityQueue.push((s, actions), 0)
    while not mypriorityQueue.isEmpty():
        state = mypriorityQueue.pop()
        if state[0] in visited_node:
            continue
        visited_node.append(state[0])
        actions = state[1]
        if (problem.isGoalState(state[0])):
            return actions
        for successor in problem.getSuccessors(state[0]):
            child_state = successor[0]
            action = successor[1]
            sub_action = list(actions)
            if not child_state in visited_node:
                sub_action.append(action)
                mypriorityQueue.push((child_state,
sub_action),heuristic(child_state, problem)
                                             + problem.getCostOfActions(sub_action))
    return actions
    util.raiseNotDefined()
```

在写完这四个函数后，在终端中输入 `python2 .\autograder.py`



```
PS D:\AI\Subject\Pacman\search> python2 .\autograder.py
Starting on 10-22 at 15:23:58
```

等到一段时间后，可以得到测试结果



```
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
```

测试通过！

五.附加实验

在完成了search实验的四个搜索算法的基础上，进一步完成附加实验，包括Corners Problem: Representation，Corners Problem: Heuristic，Eating All The Dots: Heuristic，Suboptimal Search.

1.Corner Problem

本关主要是实现一个找到所有角落的启发式函数，实现思路为先获取地图的大小，然后记录每个后继的坐标，然后通过比较判断是否为地图角落，实现代码如下

```
    def __init__(self, startingGameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height-2, self.walls.width-2
        self.corners = ((1,1), (1,top), (right, 1), (right, top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print 'Warning: no food in corner ' + str(corner)
        self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
        # Please add any code here which you would like to use
        # in initializing the problem
        "*** YOUR CODE HERE ***"
        self.top = top
        self.right = right
```

```
    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        startState = (self.startingPosition,[])
        return startState
        util.raiseNotDefined()
```

```
    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        result = state[1]
        if state[0] in self.corners:
            if state[0] not in result:
                result.append(state[0])
        if (len(result) == 4):
            return True
        else:
            return False
        util.raiseNotDefined()
```

```
    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]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            visited = list(state[1])
            if not self.walls[nextx][nexty]:
                nextState = (nextx, nexty)
                cost = 1
                if nextState in self.corners and nextState not in visited:
                    visited.append(nextState)
                successors.append(((nextState, visited), action, cost))
        self._expanded += 1 # DO NOT CHANGE
        return successors
```

2.Corners Heuristic

本关实现一个走到所有角落的最短路径的启发式函数，为之后的搜索作准备，实现思路为通过last记录四个角落，然后依次计算所有后继至四个角落的曼哈顿距离，求和之后取最小值，该后继作为下一步，实现代码如下

```
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 ***"
    visited = state[1]
    now_state = state[0]
    Heuristic = 0
    last = []
    if (problem.isGoalState(state)):
        return 0
    for i in corners:
        if i not in visited:
            last.append(i)
    pos = now_state
    cost = 999999
    while len(last) != 0:
        for i in last:
            if cost > (abs(pos[0] - i[0]) + abs(pos[1] - i[1])):
                min_con = i
                cost = (abs(pos[0] - i[0]) + abs(pos[1] - i[1]))
        Heuristic += cost
        pos = min_con
        cost = 999999
        last.remove(min_con)
    return Heuristic
```

3.Food Heuristic

本关实现一个新的搜索食物的启发式函数，使找到所有食物的路径最小。实现思路为先找到代价最大的食物，然后根据当前位置与最大代价的食物的相对位置，计算出已经吃掉的食物数量，先将代价小的食物都吃掉，最后再吃代价大的食物，即为最短路径。实现代码如下，

```
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 ***"
    lsFoodGrid = foodGrid.asList()
    last = list(lsFoodGrid)
    Heuristic = 0
    cost = 0
    max_con = position
    for i in last:
        if cost < (abs(position[0] - i[0]) + abs(position[1] - i[1])):
            max_con = i
            cost = (abs(position[0] - i[0]) + abs(position[1] - i[1]))
    Heuristic = cost
    diff = position[0] - max_con[0]
    count = 0
    for i in last:
        if diff > 0:
            if position[0] < i[0]:
                count += 1
        if diff < 0:
            if position[0] > i[0]:
                count += 1
        if diff == 0:
            if position[0] != i[0]:
                count += 1
    return Heuristic + count
```

4.Suboptimal Search

本关实现一个次优搜索，在保证吃掉所有食物的条件下，在较短的时间内找到一个比较好的路径。实现思路为使用优先队列和AnyFoodSearchProblem类中实现的函数完成本关，实现代码如下，

```
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 ***"
        Result = []
        Visited = []
        Queue = util.PriorityQueue()
        startState = (problem.getStartState(),[],0)
        Queue.push(startState,startState[2])
        while not Queue.isEmpty():
            (state,path,cost) = Queue.pop()
            if problem.isGoalState(state):
                Result = path
                break
            if state not in Visited:
                Visited.append(state)
                for currentState,currentPath,currentCost in
problem.getSuccessors(state):
                    newPath = path + [currentPath]
                    newCost = cost + currentCost
                    newState = (currentState,newPath,newCost)
                    Queue.push(newState,newCost)
        return Result
        util.raiseNotDefined()

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 ***"

        foodGrid = self.food
        if foodGrid[x][y] == True or foodGrid.count() == 0:
            return True
        else:
            return False
        util.raiseNotDefined()



def mazeDistance(point1, point2, gameState):
    """
    Returns the maze distance between any two points, using the search functions
    you have already built. The gameState can be any game state -- Pacman's
    position in that state is ignored.

    Example usage: mazeDistance( (2,4), (5,6), gameState)

    This might be a useful helper function for your ApproximateSearchAgent.
    """
    x1, y1 = point1
    x2, y2 = point2
    walls = gameState.getWalls()
    assert not walls[x1][y1], 'point1 is a wall: ' + str(point1)
    assert not walls[x2][y2], 'point2 is a wall: ' + str(point2)
    prob = PositionSearchProblem(gameState, start=point1, goal=point2,
warn=False, visualize=False)
    return len(search.bfs(prob))
```

六.实验总结

学会了四个经典的搜索算法的用法，在前期的算法课中虽然学习过DFS，BFS，aStar等搜索算法，但是之前的考试都考背诵代码，都没有真正的理解这些算法的过程。Pacman实验将算法的过程通过吃豆人的移动与路径的选择表现出来，让人眼前一亮，也让人更加深刻的理解这些算法。