1. Introduction

1.1 Background

1. Since the concept of artificial intelligence has been proposed, the development of artificial intelligence has received a lot of attention and has made great progress. It can be seen that AI has become a wide-ranging crossover and cutting-edge science. Up to now, weak AI has made great progress, while strong AI is temporarily at the bottleneck.
2. Pacman lives in a shiny blue world of twisting corridors and tasty round treats. The purpose of the game is to control Pacman to eat all the beans hidden in the maze without being caught by the ghost. Navigating this world efficiently will be Pacman's first step in mastering his domain.
3. After having AI course for several weeks, we have learned the basic knowledge of AI. In this project, we will use AI technology to control the actions of Pacman on the basis of what we have learned.

1.2 Methods

1. In this project, the Pacman agent will find a path through a maze to a required position and collecting food efficiently. We will edit files **search.py** and **searchAgents.py** and code a series of Pacman programs to complete the requirement of this project.
2. Questions that we will solve to complete the Pacman game:

- Q1: Depth First Search
- Q2: Breadth First Search
- Q3: Uniform Cost Search
- Q4: A* Search
- Q5: Corners Problem: Representation
- Q6: Corners Problem: Heuristic

1.3 Purpose

1. Complete the project.
2. Review what we have learned by this project
3. Review the utilization of Python

2.The Introduction and Implementation of Algorithms

2.1Use DFS to Find a Fixed Food Dot

This problem requires us to use DFS algorithm to plan out a path through Pacman's world and then let it execute that path step-by-step.

Depth-first search, starting from a node **v** in the graph:

1. Visit node **v**;
2. Explore as far as possible along each branch before backtracking.
3. Repeat 1) and 2) before every node in the graph is visited.

Basing on the pseudocode given in the slides, we can have our search algorithm. In this experiment, we use the structure of stack (First in Last out) to keep descendant nodes.

We implement the DFS algorithm in the **depthFirstSearch** function in search.py The code is as below:

```
def depthFirstSearch(problem):
    #初始状态
    s = problem.getStartState()
    #标记已经搜索过的状态集合exstates
    exstates = []
    #用栈实现dfs
    states = util.Stack()
    states.push((s, []))
    #循环终止条件：遍历完毕/目标测试成功
    while not states.isEmpty() and not problem.isGoalState(s):
        state, actions = states.pop()
        exstates.append(state)
        successor = problem.getSuccessors(state)
        for node in successor:
            coordinates = node[0]
            direction = node[1]
            #判断状态是否重复
            if not coordinates in exstates:
                states.push((coordinates, actions + [direction]))
            #把最后搜索的状态赋值到s，以便目标测试
            s = coordinates
    #返回动作序列
    return actions + [direction]
    util.raiseNotDefined()
```

2.2 Breadth First Search

We use **breadthFirstSearch** function to achieve breadth first search algorithm to find out a path.

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It uses the opposite strategy as depth-first search, which instead explores the highest-depth nodes first before being forced to backtrack and expand shallower nodes.

It can be seen that, in BFS and DFS, we use the same graph search algorithm but concerned data structures are different. We use stack (FILO) in **depthFirstSearch** function, while in **breadthFirtstSearch** we use queue (FIFO), because the differences between these two data structures are the key point to complete functions.

We implement the BFS algorithm in the search.py. The code as below:

```
def breadthFirstSearch(problem):
    #初始状态
    s = problem.getStartState()
    #标记已经搜索过的状态集合exstates
```

```
    exstates = []
    #用队列queue实现bfs
    states = util.Queue()
    states.push((s, []))
    while not states.isEmpty():
        state, action = states.pop()
        #目标测试
        if problem.isGoalState(state):
            return action
        #检查重复
        if state not in exstates:
            successor = problem.getSuccessors(state)
            exstates.append(state)
            #把后继节点加入队列
            for node in successor:
                coordinates = node[0]
                direction = node[1]
                if coordinates not in exstates:
                    states.push((coordinates, action + [direction]))
    #返回动作序列
    return action
    util.raiseNotDefined()
```

2.3 Varying the Cost Function

While BFS will find a fewest-actions path to the goal, we might want to find paths that are “best” in other senses. By changing the cost function (for example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas), we can encourage Pacman to find different paths.

In this experiment, we implement the uniform-cost graph search algorithm in the **uniformCostSearch** function. Uniform cost search again demands the use of a priority queue. Recall that depth first search used a priority queue with the depth upto a particular node being the priority and the path from the root to the node being the element stored. The priority queue used here is similar with the priority being the cumulative cost upto the node. Unlike depth first search where the maximum depth had the maximum priority, uniform cost search gives the minimum cumulative cost the maximum priority. Uniform cost search algorithm, in fact, is a kind of greedy algorithm. In **uniformCostSerach** function we calculate the total cost of each path, and use it as the priority to search.

The code as below:

```
def uniformCostSearch(problem):
    #初始状态
    start = problem.getStartState()
    #标记已经搜索过的状态集合exstates
    exstates = []
    #用优先队列PriorityQueue实现ucs
    states = util.PriorityQueue()
    states.push((start, []) ,0)
    while not states.isEmpty():
        state, actions = states.pop()
        #目标测试
```

```
        if problem.isGoalState(state):
            return actions
        #检查重复
        if state not in exstates:
            #扩展
            successors = problem.getSuccessors(state)
            for node in successors:
                coordinate = node[0]
                direction = node[1]
                if coordinate not in exstates:
                    newActions = actions + [direction]
                    #ucs比bfs的区别在于getCostOfActions决定节点扩展的优先级
                    states.push((coordinate, actions + [direction]),
problem.getCostOfActions(newActions))
        exstates.append(state)
    return actions
    util.raiseNotDefined()
```

2.4 A* Search

The most widely known form of best-first search is called A∗ search. It evaluates nodes by combining **g(n)**, the cost to reach the node, and **h(n)**, the cost to get from the node to the goal:

f(n) = **g(n)** +**h(n)**。

Since **g(n)** gives the path cost from the start node to node n, and **h(n)** is the estimated cost of the cheapest path from n to the goal, we have

f(n) = estimated cost of the cheapest solution through n.

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the

node with the lowest value of **g(n)** + **h(n)**. The algorithm is identical to the uniform cost search except that A* uses **g(n)** + h instead of **g**.

In this experiment, we use manhattan distance as the heuristic function.

We use *PriorityQueuewithFunction* as the data structure.

The code as below:

```
def aStarSearch(problem, heuristic=nullHeuristic):
    "Search the node that has the lowest combined cost f(n) and heuristic g(n)
first."
    start = problem.getStartState()
    exstates = []
    # 使用优先队列，每次扩展都是选择当前代价最小的方向
    states = util.PriorityQueue()
    states.push((start, []), nullHeuristic(start, problem))
    nCost = 0
    while not states.isEmpty():
        state, actions = states.pop()
        #目标测试
        if problem.isGoalState(state):
            return actions
```

```
        #检查重复
        if state not in exstates:
            #扩展
            successors = problem.getSuccessors(state)
            for node in successors:
                coordinate = node[0]
                direction = node[1]
                if coordinate not in exstates:
                    newActions = actions + [direction]
                    #计算动作代价和启发式函数值得和
                    newCost = problem.getCostOfActions(newActions) +
heuristic(coordinate, problem)
                    states.push((coordinate, actions + [direction]), newCost)
        exstates.append(state)
    #返回动作序列
    return actions
    util.raiseNotDefined()
```

2.5 Corners Problem: Representation

In corner mazes, there are four dots, one in each cornet. The search problem is to find the shortest path through the maze with touching all four corners. And it is required that we should not encode irrelevant information. The key point is how to represent the state of Pacman, it is apparent the location cannot be the state solely.

We use ***int*** function to store the walls, Pacman's starting position and corners in **CornersProblem** class. And we define **getStartState** function to acquire the start state, **isGoalState** function to return whether this search state is a goal state of the problem, **getSuccessors** function to return successor states, the actions they require, and a cost of 1, and **getCostOfActions** function to return the cost of a particular sequence of actions.

When finding out descent nodes, traversing four directions and using **directionToVector** to move. If there is no wall, appending the next position, action, and cost to the next code.

We implement **getStartState** function as following:

```
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.
    You must select a suitable state space and successor function
    """

    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 # Number of search nodes expanded
        # Please add any code here which you would like to use
        # in initializing the problem
        "*** YOUR CODE HERE ***"
        self.right = right
        self.top = top

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        #初始节点（开始位置，角落情况）
        allCorners = (False, False, False, False)
        start = (self.startingPosition, allCorners)
        return start
        util.raiseNotDefined()

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        #目标测试：四个角落都访问过
        corners = state[1]
        boolean = corners[0] and corners[1] and corners[2] and corners[3]
        return boolean
        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 = []
        #遍历能够做的后续动作
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:
            # Add a successor state to the successor list if the action is legal
            "*** YOUR CODE HERE ***"
             #   x,y = currentPosition
            x,y = state[0]
            holdCorners = state[1]
            #   dx, dy = Actions.directionToVector(action)
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            newCorners = ()
            nextState = (nextx, nexty)
```

```
            #不碰墙
            if not hitsWall:
                #能到达角落，四种情况判断
                if nextState in self.corners:
                    if nextState == (self.right, 1):
                        newCorners = [True, holdCorners[1], holdCorners[2],
holdCorners[3]]
                    elif nextState == (self.right, self.top):
                        newCorners = [holdCorners[0], True, holdCorners[2],
holdCorners[3]]
                    elif nextState == (1, self.top):
                        newCorners = [holdCorners[0], holdCorners[1], True,
holdCorners[3]]
                    elif nextState == (1,1):
                        newCorners = [holdCorners[0], holdCorners[1],
holdCorners[2], True]
                    successor = ((nextState, newCorners), action,  1)
                #去角落的中途
                else:
                    successor = ((nextState, holdCorners), action, 1)
                successors.append(successor)
        self._expanded += 1
        return successors

    def getCostOfActions(self, actions):
        """
        Returns the cost of a particular sequence of actions.  If those actions
        include an illegal move, return 999999.  This is implemented for you.
        """
        if actions == None: return 999999
        x,y= self.startingPosition
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]: return 999999
        return len(actions)
```

2.6 Corners Problem: Heuristic

Finding out a non-trivial and consistent heuristic function to solve the corners problem with fewer nodes as possible. The core idea of our heuristic is to use the sum of four manhattan distances between current position and each corner as heuristic cost, through which our heuristic function----**cornersHeuristic---**can be defined.

Heuristic function as following:

1. current state is the starting point.
2. If the corners are not fully traversed, the Manhattan distance between the current node and the traversing corners is calculated, and the minimum distance is calculated.
3. Select the smallest distance corner and add the smallest distance to hn.
4. Re-use the minimum distance corner as the starting point, and remove it from the corners.
5. Go to 2.

The code as following:

```python
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 ***"
    position = state[0]
    stateCorners = state[1]
    corners = problem.corners
    top = problem.walls.height-2
    right = problem.walls.width-2
    node = []
    for c in corners:
        if c == (1,1):
            if not stateCorners[3]:
                node.append(c)
        if c == (1, top):
            if not stateCorners[2]:
                node.append(c)
        if c == (right, top):
            if not stateCorners[1]:
                node.append(c)
        if c == (right, 1):
            if not stateCorners[0]:
                node.append(c)
    cost = 0
    currPosition = position
    while len(node) > 0:
        distArr= []
        for i in range(0, len(node)):
            dist = util.manhattanDistance(currPosition, node[i])
            distArr.append(dist)
        mindist = min(distArr)
        cost += mindist
        minDistI= distArr.index(mindist)
        currPosition = node[minDistI]
        del node[minDistI]
    return cost
```

Proof the admissible and consistent nature.

1. The distance moving on the chessboard is always greater than or equal to the distance of Manhattan. Pacman can only move up and down, left and right. So the heuristic function cost is less than or equal to the reality path cost.

2. This heuristic is consistent. For each node, the H value of its successor node plus the distance consumption value of the current node to the successor node is greater than or equal to the H value of the current node. Since the actual value from one corner to any other corner is greater than the Manhattan distance, and the successor node arrives at the nearest corner and the current node arrives at the nearest corner can become consistent according to the symmetric transformation.

3.The Results

3.1 Depth First Search

python pacman.py -l tinyMaze -p SearchAgent

the result are as follow:

```
(python2.7) PS D:\code_learning\Blog\pacman\search> python pacman.py -l tinyMaze
-p SearchAgent
[SearchAgent] using function uniformCostSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```

python pacman.py -l mediumMaze -p SearchAgent

```
(python2.7) PS D:\code_learning\Blog\pacman\search> python pacman.py -l
mediumMaze -p SearchAgent
[SearchAgent] using function uniformCostSearch
[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
```

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

```
(python2.7) PS D:\code_learning\Blog\pacman\search> python pacman.py -l bigMaze -
z .5 -p SearchAgent
[SearchAgent] using function uniformCostSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 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
```

3.2 Breadth First Search

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

```
(python2.7) PS D:\code_learning\Blog\pacman\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
```

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

```
(python2.7) PS D:\code_learning\Blog\pacman\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
```

3.3 Varyring the Cost Function

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

```
python2.7) PS D:\code_learning\Blog\pacman\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
```

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

```
(python2.7) PS D:\code_learning\Blog\pacman\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
```

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

```
(python2.7) PS D:\code_learning\Blog\pacman\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
```

3.4 A* Search

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

```
 (python2.7) PS D:\code_learning\Blog\pacman\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 0.1 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
```

3.5 Corners Problem: Representation

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

```
 (python2.7) PS D:\code_learning\Blog\pacman\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
```

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

```
 (python2.7) PS D:\code_learning\Blog\pacman\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 0.2 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
```

3.6 Corners Problem: Heuristic

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

```
(python2.7) PS D:\code_learning\Blog\pacman\search> python pacman.py -l
mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.1 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
```

3.7 Auto-grade

Results of all code tests:

python autograder.py

```
Provisional grades
==================

Question q1: 8/8
Question q2: 8/8
Question q3: 8/8
Question q4: 8/8
Question q5: 10/10

Question q6: 8/8
------------------

Total: 50/50
```

4.Discussion

1. Through this project, we realize BFS, DFS, uniform cost search, and A* search by using Python. It is undoubted that we learn more about search algorithms of AI. Compared to BFS as well as DFS, uniform cost search is more efficient. However, if the condition is not suitable for using uniform cost search, it equals BFS. Overall, A* search is the most efficient search algorithm among these four algorithms because it can ignore many unnecessary paths by pre-treating the function.
2. For heuristics, probably the most important thing is the heuristic function---**h(n)**. If the estimate function **h(n)** is accurate and reliable enough, the process of search is promoted a lot generally.
3. This project is to complete the code to satisfy the requirement of the experiment, basing on the given code. During the project, we should not only complete our own code but also read the given code written by others, which is definitely helpful.