CS210 : ARTIFICIAL INTELLIGENCE LAB

LAB ASSIGNMENT 4_5: AI & Python

Submitted By:

VAIBHAV GUPTA

AKSHAT SAHU

Roll No: U22CS029 AND U22CS034

Branch: CSE

Semester: 4th Sem

Division : A

Submitted To: Dr. Chandra Prakash

Department of Computer Science and Engineering

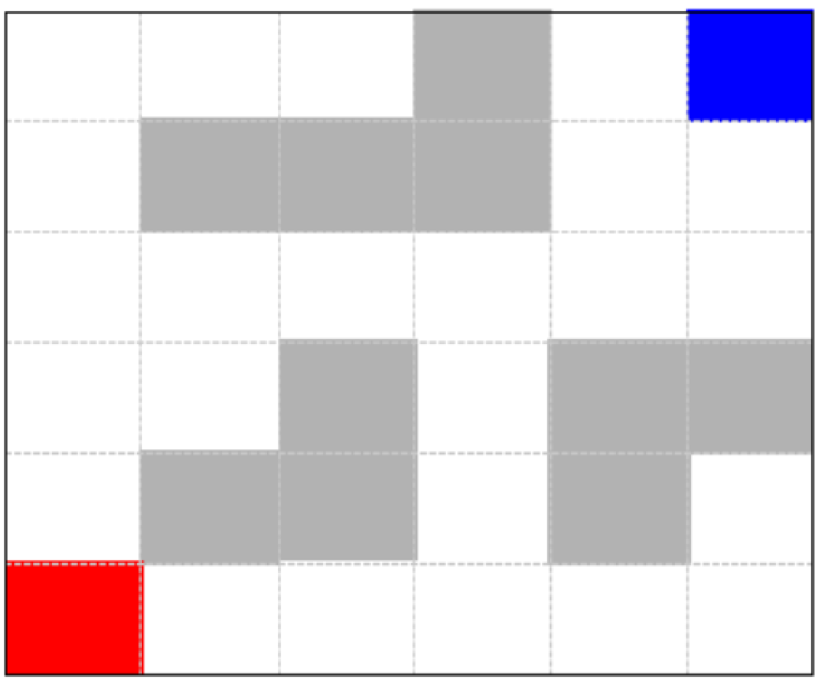
SV NATIONAL INSTITUTE OF TECHNOLOGY

SURAT

2024

PART A : Introductory Problem [25 Marks]
Maze Problem [15 Marks]

1. Consider a maze comprising of square blocks in which intelligent agent can move either vertically or
horizontally. Diagonal movement is not allowed. Cost of each move is 1.



Red block: initial position, **Blue block:** goal position and **Grey block:** obstacle
Apply following Blind/Uninformed and Informed algorithms :
(a) dfs: Depth first search
(b) bfs: Breadth first search
(c) dls: Depth limited search, use 3 as default depth
(d) ucs: Uniform cost Search
(e) gbfs: Greedy Best First Search
(f) astar: A* Algorithm
Inputs:

Write a python program that takes input number of square blocks as input (i.e. 6 x 6) in first line.
Second line contains the initial position of intelligent agent which is (1,1) and the goal square block
which is (6,6) in above example. Third line contains the coordinates of the obstacles. Fourth line
contains the search strategy.
eg. input file: input.txt
6,6
1-1
1,1;6,6
2,1;2,5;3,2;3,3;3,5;4,5;4,6;5,2;5,3;6,3
dfs
python TA_4_5_P2_maze_world.py “input2.*txt*ȷı*output*2.*txt*ȷ
Outputs: Sequence of blocks that are explored (each on separate line) as per search algorithm so as to
reach goal position. Last line should contain the total search cost.

CODE

```
import heapq

class Maze:
    def __init__(self, size, start, goal, obstacles):
        self.size = size
        self.start = start
        self.goal = goal
        self.obstacles = obstacles

    def is_valid_move(self, position):
        x, y = position
        return 1 <= x <= self.size[0] and 1 <= y <= self.size[1] and position not in
self.obstacles

    def get_neighbors(self, position):
        x, y = position
        neighbors = [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]
        return [neighbor for neighbor in neighbors if self.is_valid_move(neighbor)]

def dfs(maze):
    stack = [(maze.start, [maze.start])]
    visited = set()
```

```python
    while stack:
        current, path = stack.pop()
        if current == maze.goal:
            return path
        if current not in visited:
            visited.add(current)
            for neighbor in maze.get_neighbors(current):
                stack.append((neighbor, path + [neighbor]))
    return None

def bfs(maze):
    queue = [(maze.start, [maze.start])]
    visited = set()

    while queue:
        current, path = queue.pop(0)
        if current == maze.goal:
            return path
        if current not in visited:
            visited.add(current)
            for neighbor in maze.get_neighbors(current):
                queue.append((neighbor, path + [neighbor]))
    return None

def dls(maze, depth_limit):
    stack = [(maze.start, [maze.start], 0)]
    visited = set()

    while stack:
        current, path, depth = stack.pop()
        if current == maze.goal:
            return path
        if depth < depth_limit and current not in visited:
            visited.add(current)
            for neighbor in maze.get_neighbors(current):
                stack.append((neighbor, path + [neighbor], depth + 1))
    return None

def ucs(maze):
    # Uniform Cost Search using heapq
    heap = [(0, maze.start, [maze.start])]
    visited = set()

    while heap:
        cost, current, path = heapq.heappop(heap)
        if current == maze.goal:
            return path
        if current not in visited:
            visited.add(current)
            for neighbor in maze.get_neighbors(current):
                new_cost = cost + 1  # Assuming cost of each move is 1
                heapq.heappush(heap, (new_cost, neighbor, path + [neighbor]))
    return None

def gbfs(maze):
    # Greedy Best First Search
```

```
    queue = [(heuristic(maze.start, maze.goal), maze.start, [maze.start])]
    visited = set()

    while queue:
        _, current, path = heapq.heappop(queue)
        if current == maze.goal:
            return path
        if current not in visited:
            visited.add(current)
            for neighbor in maze.get_neighbors(current):
                heapq.heappush(queue, (heuristic(neighbor, maze.goal), neighbor, path
+ [neighbor]))
    return None

def astar(maze):

    queue = [(heuristic(maze.start, maze.goal), 0, maze.start, [maze.start])]
    visited = set()

    while queue:
        _, cost, current, path = heapq.heappop(queue)
        if current == maze.goal:
            return path
        if current not in visited:
            visited.add(current)
            for neighbor in maze.get_neighbors(current):
                new_cost = cost + 1  # Assuming cost of each move is 1
                heapq.heappush(queue, (new_cost + heuristic(neighbor, maze.goal),
new_cost, neighbor, path + [neighbor]))
    return None


def heuristic(current, goal):
    return abs(current[0] - goal[0]) + abs(current[1] - goal[1])


def read_input(file_path):
    with open(file_path, 'r') as file:
        size = tuple(map(int, file.readline().strip().split(',')))
        start, goal = map(tuple, [map(int, coord.split(',')) for coord in
file.readline().strip().split(';')])
        obstacles = [tuple(map(int, coord.split(','))) for coord in
file.readline().strip().split(';')]
        method = file.readline().strip()
    return size, start, goal, obstacles, method


def main():
    size, start, goal, obstacles, method = read_input("input.txt")
    maze = Maze(size, start, goal, obstacles)

    if method == "dfs":
        print("Approach Defth First Search :  \nBlocks Travelled ", dfs(maze))
    elif method == "bfs":
        print("Approach Breadth First Search :  \nBlocks Travelled ",bfs(maze))
    elif method == "dls":
```

```
        print("Approach Depth Limit Search(upto 3) :  \nBlocks Travelled ",dls(maze,
depth_limit=3))
    elif method == "ucs":
        print("Approach Uniform Cost Search :  \nBlocks Travelled ",ucs(maze))
    elif method == "gbfs":
        print("Approach Greedy Breadth First Search :  \nBlocks Travelled
",gbfs(maze))
    elif method == "astar":
        print("Approach Astar :  \nBlocks Travelled ",astar(maze))
    else:
        print("Invalid search strategy.")

if __name__ == "__main__":
    main()
```

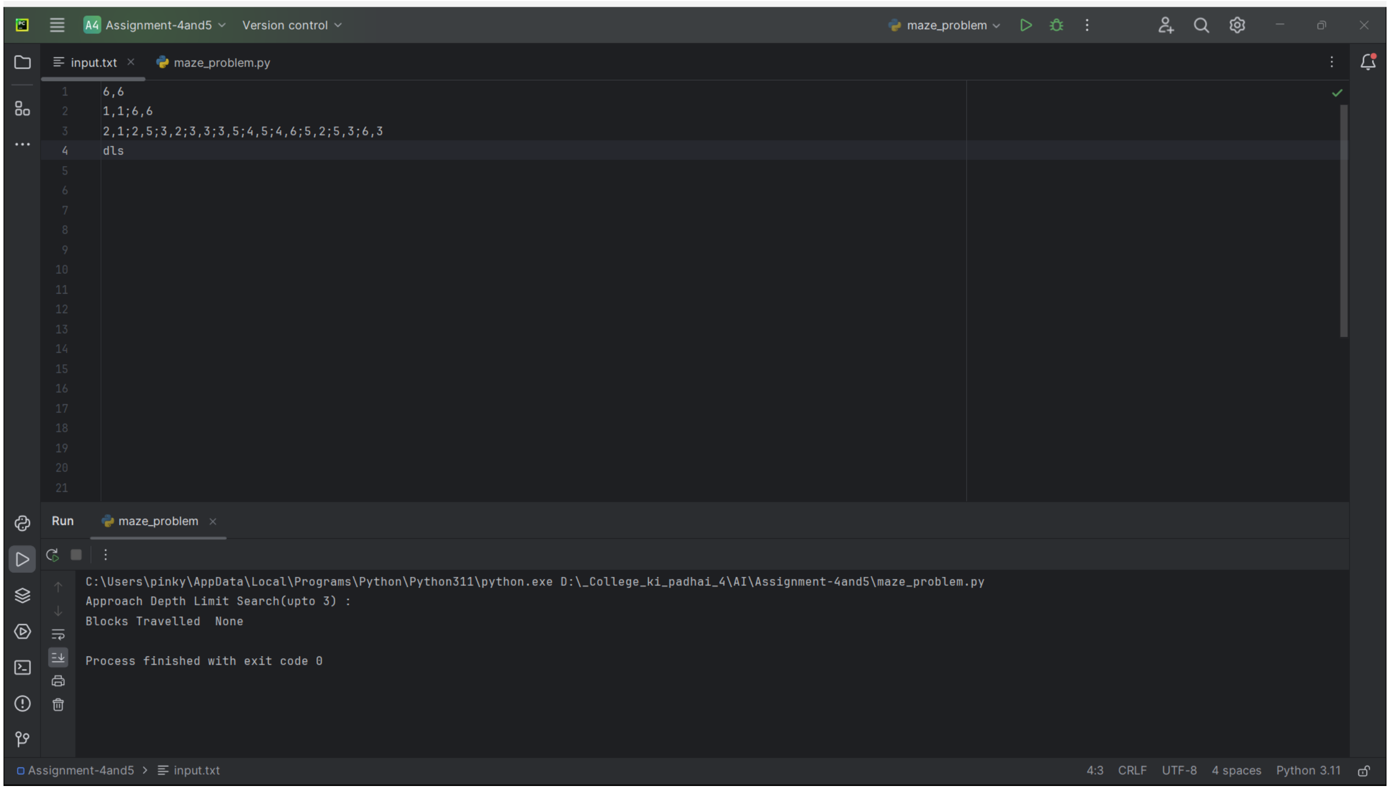
OUTPUT

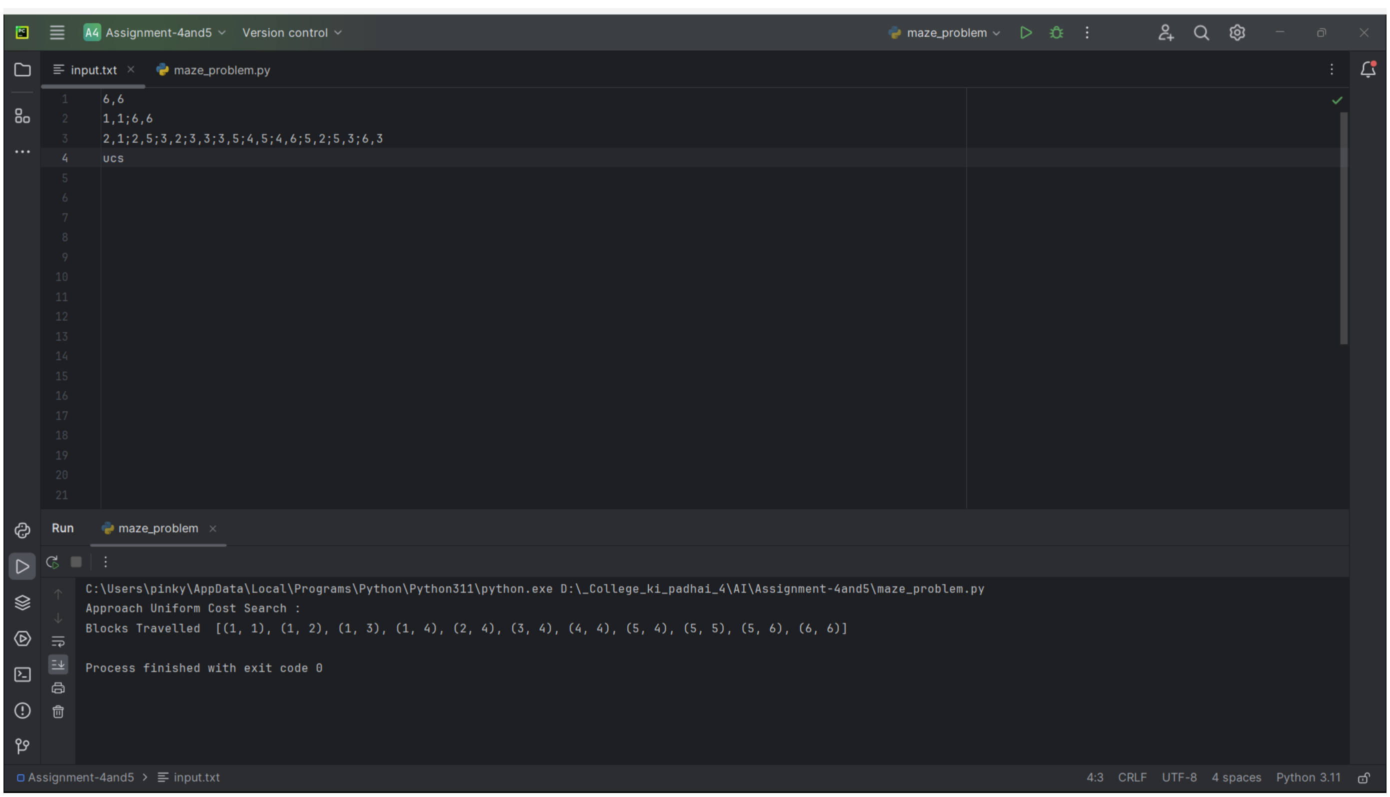
```
Approach Defth First Search :
Blocks Travelled  [(1, 1), (1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4), (5, 5), (5, 6), (6, 6)]

Process finished with exit code 0
```

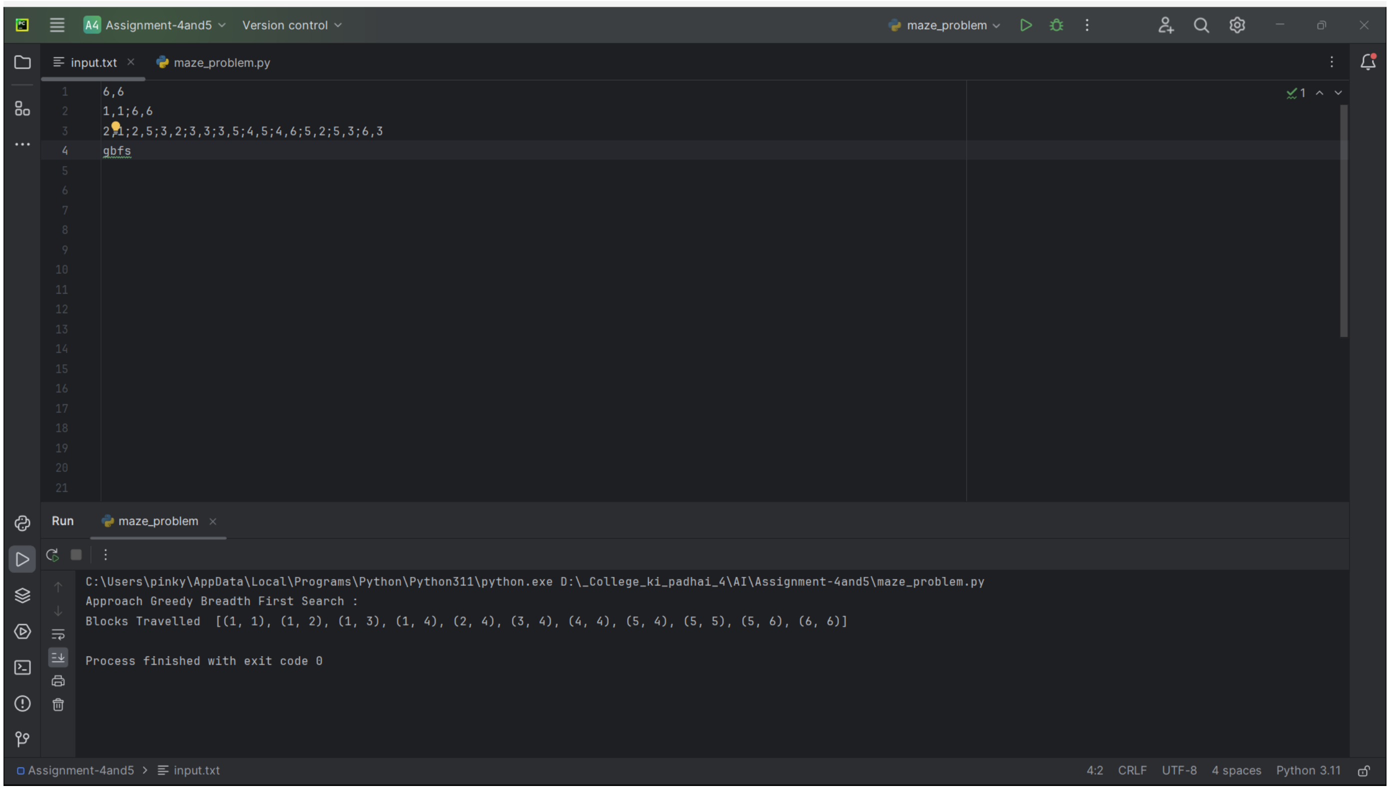
```
input.txt   maze_problem.py

6,6
1,1;6,6
2,1;2,5;3,2;3,3;3,5;4,5;4,6;5,2;5,3;6,3
dfs
```

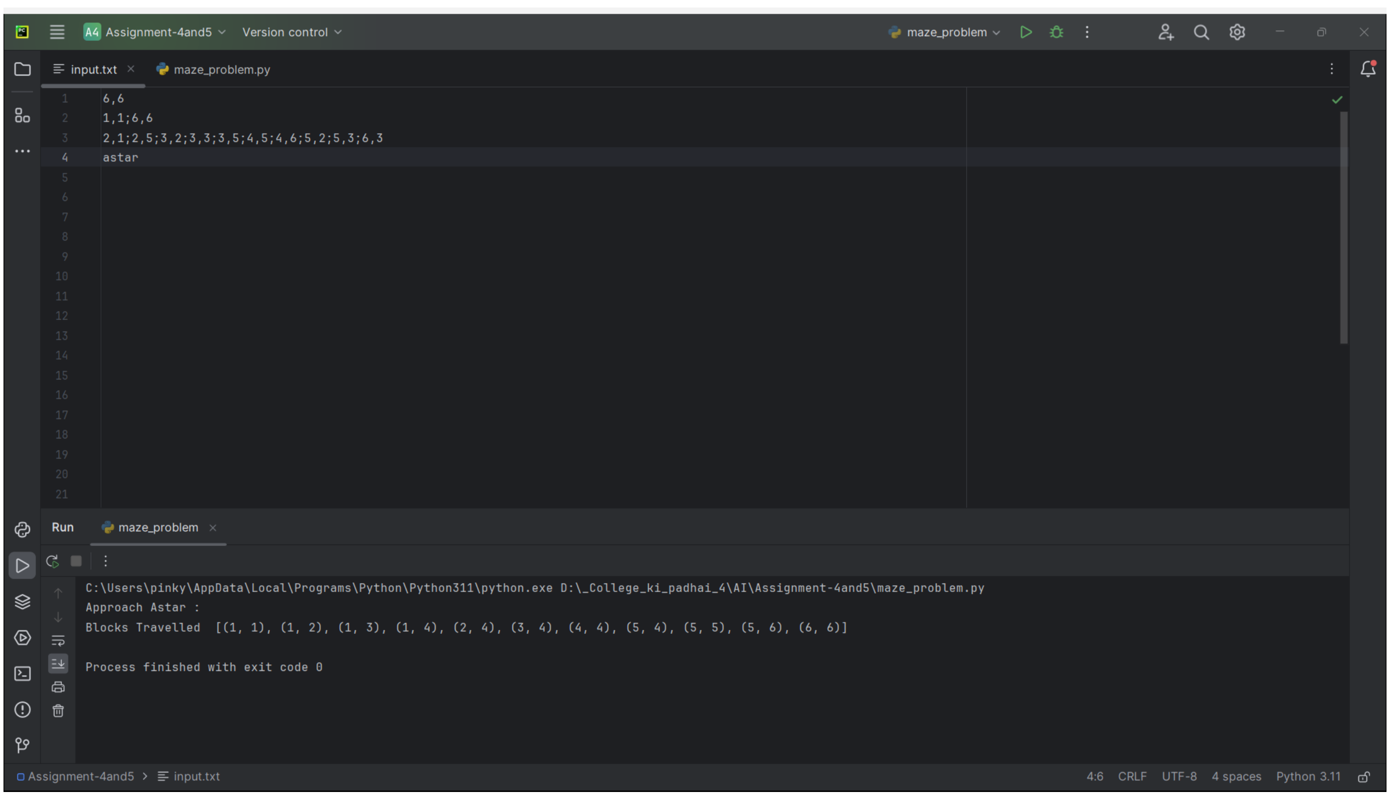
```
Approach Breadth First Search :
Blocks Travelled  [(1, 1), (1, 2), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4), (5, 4), (6, 4), (6, 5), (6, 6)]

Process finished with exit code 0
```

```
input.txt   maze_problem.py

6,6
1,1;6,6
2,1;2,5;3,2;3,3;3,5;4,5;4,6;5,2;5,3;6,3
bfs
```

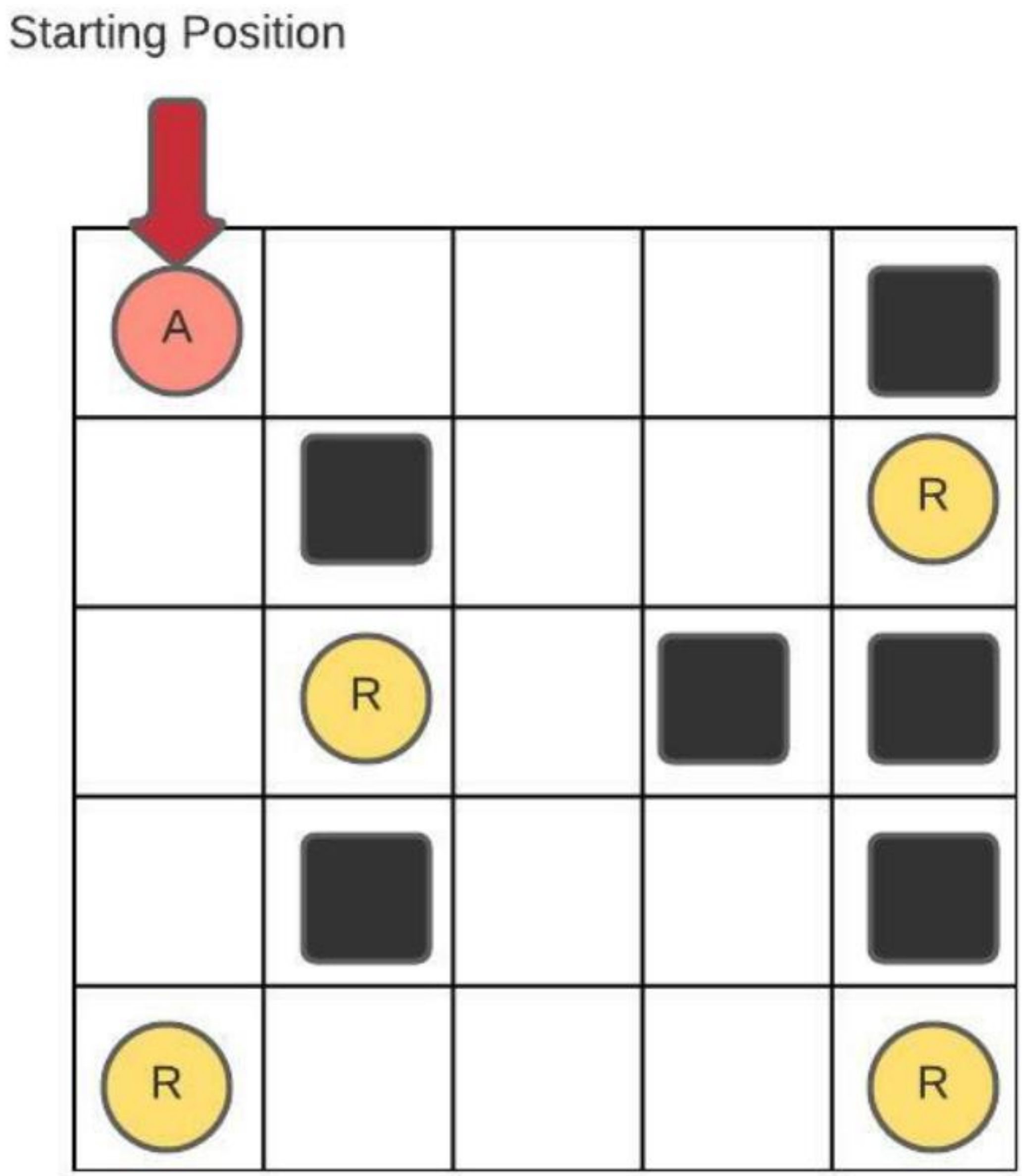

Assignment-4and5
Version control
maze_problem
input.txt
maze_problem.py
6,6
1,1;6,6
2,1;2,5;3,2;3,3;3,5;4,5;4,6;5,2;5,3;6,3
dls
Run
maze_problem
C:\Users\pinky\AppData\Local\Programs\Python\Python311\python.exe D:_College_ki_padhai_4\AI\Assignment-4and5\maze_problem.py
Approach Depth Limit Search(upto 3) :
Blocks Travelled None
Process finished with exit code 0
Assignment-4and5 > input.txt
4:3 CRLF UTF-8 4 spaces Python 3.11


Assignment-4and5
Version control
maze_problem
input.txt
maze_problem.py
6,6
1,1;6,6
2,1;2,5;3,2;3,3;3,5;4,5;4,6;5,2;5,3;6,3
ucs
Run
maze_problem
C:\Users\pinky\AppData\Local\Programs\Python\Python311\python.exe D:_College_ki_padhai_4\AI\Assignment-4and5\maze_problem.py
Approach Uniform Cost Search :
Blocks Travelled [(1, 1), (1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4), (5, 5), (5, 6), (6, 6)]
Process finished with exit code 0
Assignment-4and5 > input.txt
4:3 CRLF UTF-8 4 spaces Python 3.11


Assignment-4and5
Version control
maze_problem
input.txt
maze_problem.py
6,6
1,1;6,6
2,1;2,5;3,2;3,3;3,5;4,5;4,6;5,2;5,3;6,3
gbfs
Run
maze_problem
C:\Users\pinky\AppData\Local\Programs\Python\Python311\python.exe D:_College_ki_padhai_4\AI\Assignment-4and5\maze_problem.py
Approach Greedy Breadth First Search :
Blocks Travelled [(1, 1), (1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4), (5, 5), (5, 6), (6, 6)]
Process finished with exit code 0
Assignment-4and5 > input.txt
4:2 CRLF UTF-8 4 spaces Python 3.11


Assignment-4and5
Version control
maze_problem
input.txt
maze_problem.py
6,6
1,1;6,6
2,1;2,5;3,2;3,3;3,5;4,5;4,6;5,2;5,3;6,3
astar
Run
maze_problem
C:\Users\pinky\AppData\Local\Programs\Python\Python311\python.exe D:_College_ki_padhai_4\AI\Assignment-4and5\maze_problem.py
Approach Astar :
Blocks Travelled [(1, 1), (1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4), (5, 5), (5, 6), (6, 6)]
Process finished with exit code 0
Assignment-4and5 > input.txt
4:6 CRLF UTF-8 4 spaces Python 3.11

2. **Maze problem with multi-goal [10 Marks]**
Consider the maze given in the figure below. The walled tiles are marked in black and your agent A cannot move to or through those positions.



Inputs:
Write python/C program that takes the maze as a 5x5 matrix input where 0 denotes an empty tile, 1
denotes an obstruction/wall, 2 denotes the start state and 3 denotes the reward. Assume valid actions as

L,R,U,D,S,N where L=move_left, R=move_right, U=move_up, D=move_down.
Use the **A* algorithms as (astar)** on the resultant maze for your agent to reach all the rewards, and keep a record of the tiles visited on the way.

Hints:
a) Your program should create the appropriate data structure that can capture problem states, as mentioned in the problem.
b) Once the goal is reached (i.e. Reward position), program should terminate.

Outputs: The output should have the sequence of the tiles visited by each algorithm to reach the termination
state stored in an output file labeled as "out_astar.txt" and so on. Print the number of steps required to
reach the goal.

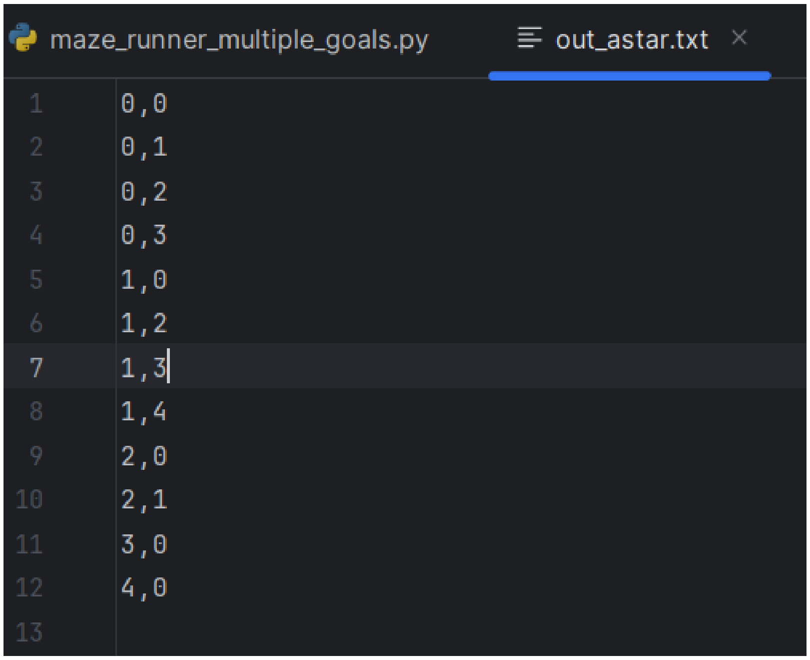
CODE

```
import heapq

class MazeSolver:
    def __init__(self, maze):
        self.maze = maze
        self.rows = len(maze)
        self.cols = len(maze[0])
        self.start = self.find_start()
        self.rewards = self.find_rewards()
        self.visited = set()

    def find_start(self):
        for i in range(self.rows):
            for j in range(self.cols):
                if self.maze[i][j] == 2:
                    return (i, j)

    def find_rewards(self):
        rewards = []
        for i in range(self.rows):
            for j in range(self.cols):
                if self.maze[i][j] == 3:
                    rewards.append((i, j))
        return rewards

    def heuristic(self, current, goal):
        return abs(current[0] - goal[0]) + abs(current[1] - goal[1])

    def is_valid_move(self, position):
```

```
        x, y = position
        return 0 <= x < self.rows and 0 <= y < self.cols and self.maze[x][y] != 1 and
position not in self.visited

    def astar(self, start, goal):
        heap = [(0, start, [])]

        while heap:
            _, current, path = heapq.heappop(heap)

            if current == goal:
                self.visited.update(path)
                return path

            if current not in self.visited:
                self.visited.add(current)

                moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

                for move in moves:
                    next_position = (current[0] + move[0], current[1] + move[1])
                    if self.is_valid_move(next_position):
                        heapq.heappush(heap, (self.heuristic(next_position, goal) +
len(path), next_position, path + [next_position]))

        return []

    def solve_maze(self):
        start = self.start
        total_steps = 0

        for reward in self.rewards:
            path = self.astar(start, reward)
            total_steps += len(path) - 1
            start = reward

        return total_steps

    def write_output(self, filename):
        with open(filename, 'w') as file:
            for position in sorted(self.visited):
                file.write(f"{position[0]},{position[1]}\n")


if __name__ == "__main__":
    maze = [
        [2, 0, 0, 0, 1],
        [0, 1, 0, 0, 3],
        [0, 3, 0, 1, 1],
        [0, 1, 0, 0, 1],
        [3, 0, 0, 0, 3]
    ]

    solver = MazeSolver(maze)
    total_steps = solver.solve_maze()
```

```
    print("Total Steps:", total_steps)
    solver.write_output("out_astar.txt")
```

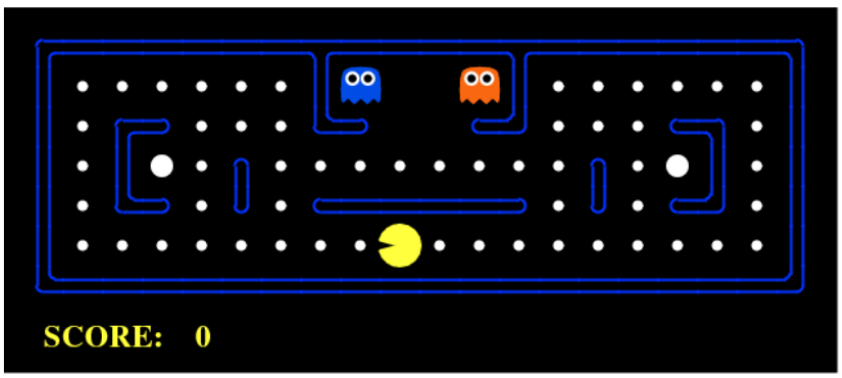
OUTPUT

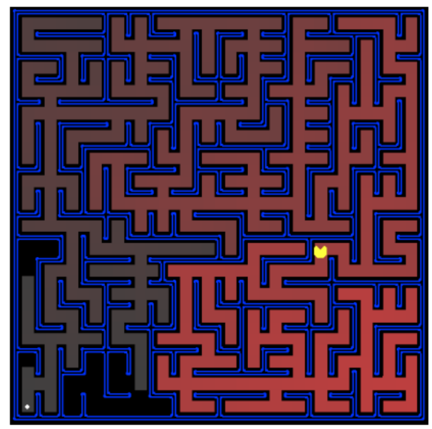

maze_runner_multiple_goals.py   out_astar.txt

```
0,0
0,1
0,2
0,3
1,0
1,2
1,3
1,4
2,0
2,1
3,0
4,0
```

PART B : Exploratory Problem [25 Marks]

3. **Search in Pac-Man** This problem allow you to visualize the results of the techniques you implement. Pac-Man provides a challenging problem environment that demands creative solutions of a real-world AI problems. The Pacman agent needs to find paths through the maze world, both to reach a location and to collect food efficiently. In this Problem, you are expected to implement and experiment with different AI search techniques that was discussed in the class in a Pacman environment.
This lab assignment is inspired by Project 1: Search, which is a part of a recent offering of CS188 at UC Berkeley[1]. We thank the authors at Berkeley for making their project available to the public.





Aim: Students implement depth-first, breadth-first, uniform cost, and A* search algorithms. These algorithms are used to solve navigation and traveling salesman problems in the Pacman world.

The code for this assignment is provided to you as LA_4_5_search zip folder. You can download all the code and supporting files as a Folder/ zip archive from the shared link. The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore.

Assumption : The projects for this class assume you use Python 3.6.

Files you'll edit:

search.py : Where all of your search algorithms will reside.
searchAgents.py : Where all of your search-based agents will reside.

Files you might want to look at:

pacman.py	The main file that runs Pacman games. This file describes a Pacman GameState
type, which you use in this project.	

game.py	The logic behind how the Pacman world works. This file describes several
supporting types like AgentState, Agent, Direction, and Grid.	
util.py	Useful data structures for implementing search algorithms.

Supporting files you can ignore:

graphicsDisplay.py	Graphics for Pacman
graphicsUtils.py	Support for Pacman graphics
textDisplay.py	ASCII graphics for Pacman

ghostAgents.py	Agents to control ghosts
keyboardAgents.py	Keyboard interfaces to control Pacman
layout.py	Code for reading layout files and storing their contents
autograder.py	Project autograder
testParser.py	Parses autograder test and solution files
testClasses.py	General autograding test classes
test_cases/ searchTestClasses.py	Directory containing the test cases for each question Project 1 specific autograding test classes

Files to Edit and Submit: You will fill in portions of search.py and searchAgents.py during the assignment. Once you have completed the assignment, you will submit a token generated by submission_autograder.py. Please *do not* change the other files in this distribution or submit any of our original files other than these files.

Welcome to Pacman

TASK 0: Understanding the Working of Pacman Game (Ungraded)

After downloading the code from the shared link, unzipping it, and changing to the directory, you should be able to :

TASK 0: Play the game of Pacman by typing the following at the command line:

python pacman.py

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

The simplest agent in searchAgents.py is called the GoWestAgent, which always goes West (a trivial reflex agent). This agent can occasionally win:

python pacman.py --layout testMaze --pacman GoWestAgent

But, things get ugly for this agent when turning is required:

python pacman.py --layout tinyMaze --pacman GoWestAgent

If Pacman gets stuck, you can exit the game by typing CTRL-c into your terminal.

Soon, your agent will solve not only tinyMaze, but any maze you want.
Note that pacman.py supports a number of options that can each be expressed in a long way (e.g., --layout) or a short way (e.g., -l). You can see the list of all options and their default values via:
python pacman.py -h
Also, all of the commands that appear in this project also appear in commands.txt, for easy copying and pasting. In UNIX/Mac OS X, you can even run all these commands in order with bash commands.txt.

Question 4.1 (3 points) : Finding a Fixed Food Dot using Depth First Search

In searchAgents.py, you'll find a fully implemented SearchAgent, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented – that's your job.

First, test that the SearchAgent is working correctly by running:
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch

The command above tells the SearchAgent to use tinyMazeSearch as its search algorithm, which is implemented in search.py. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture slides. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Important note: All of your search functions need to return a list of *actions* that will lead the agent
from the start to the goal. These actions all have to be legal moves (valid directions, no moving through
walls).

Important note: Make sure to **use** the Stack, Queue and PriorityQueue data structures provided to you in util.py! These data structure implementations have particular properties which are required for compatibility with the autograder.

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the fringe is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy. (Your implementation need *not* be of this form to receive full credit).

Implement the depth-first search (DFS) algorithm in the depthFirstSearch function in search.py. To make your algorithm *complete*, write the graph search version of DFS, which avoids expanding any already visited states.

Your code should quickly find a solution for:

python pacman.py -l tinyMaze -p SearchAgent

python pacman.py -l mediumMaze -p SearchAgent

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

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

Hint: If you use a Stack as your data structure, the solution found by your DFS algorithm for mediumMaze should have a length of 130 (provided you push successors onto the fringe in the order provided by getSuccessors; you might get 246 if you push them in the reverse order). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

```
Starting on 2-5 at 23:12:02

Question q1
===========
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'D', 'C']
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
***     solution:               ['2:A->D', '0:D->G']
***     expanded_states:        ['A', 'D']
*** PASS: test_cases\q1\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_manypaths.test
***     solution:               ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']
***     expanded_states:        ['A', 'B2', 'C', 'D', 'E2', 'F']
*** PASS: test_cases\q1\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 130
***     nodes expanded:         146

### Question q1: 3/3 ###
```

Question 4.2 (3 points): Breadth First Search

Implement the breadth-first search (BFS) algorithm in the breadthFirstSearch function in search.py. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search. python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5 Does BFS find a least cost solution? If not, check your implementation.

Hint: If Pacman moves too slowly for you, try the option --frameTime 0.

Note: If you've written your search code generically, your code should work equally well for the eightpuzzle search problem without any changes. python eightpuzzle.py

```
Question q2
===========
*** PASS: test_cases\q2\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q2\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q2\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q2\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 3/3 ###
```

Question 4.3 (3 points): 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. Consider mediumDottedMaze and mediumScaryMaze. By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rationalPacman agent should adjust its behavior in response.

Implement the uniform-cost graph search algorithm in the uniformCostSearch function in search.py. We encourage you to look through util.py for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

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

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

Note: You should get very low and very high path costs for the StayEastSearchAgent and StayWestSearchAgent respectively, due to their exponential cost functions (see searchAgents.py for details).

```
Question q3
===========
*** PASS: test_cases\q3\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q3\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q3\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q3\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q3\ucs_0_graph.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q3\ucs_1_problemC.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269
*** PASS: test_cases\q3\ucs_2_problemE.test
***     pacman layout:          mediumMaze
***     solution length: 74
***     nodes expanded:         260
*** PASS: test_cases\q3\ucs_3_problemW.test
***     pacman layout:          mediumMaze
***     solution length: 152
***     nodes expanded:         173
*** PASS: test_cases\q3\ucs_4_testSearch.test
***     pacman layout:          testSearch
***     solution length: 7
***     nodes expanded:         14
*** PASS: test_cases\q3\ucs_5_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']

### Question q3: 3/3 ###
```

Question 4.4 (3 points): A* search

Implement A* graph search in the empty function aStarSearch in search.py. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The nullHeuristic heuristic function in search.py is a trivial example. You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as manhattanHeuristic in searchAgents.py). python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic You should see that A* finds the optimal solution slightly faster than uniform cost search (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly). What happens on openMaze for the various search strategies?

```
Question q4
===========
*** PASS: test_cases\q4\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q4\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases\q4\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases\q4\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q4\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q4\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###
```

Question 4.5 (3 points): Finding All the Corners
The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.
In *corner mazes*, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not).

Note that for some mazes like tinyCorners, the shortest path does not always go to the closest food first! *Hint*: the shortest path through tinyCorners takes 28 steps.
Note: Make sure to complete Question 2 before working on Question 5, because Question 5 builds upon your answer

for Question 2.
Implement the CornersProblem search problem in searchAgents.py. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Now, your search agent should solve:

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

python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
To receive full credit, you need to define an abstract state representation that *does not* encode irrelevant information (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pacman GameState as a search state. Your code will be very, very slow if you do (and also wrong).

Hint: The only parts of the game state you need to reference in your implementation are the starting Pacman position and the location of the four corners. Our implementation of breadthFirstSearch expands just under 2000 search nodes on mediumCorners. However, heuristics (used with A* search) can reduce the amount of searching required.

```
Question q5
===========
*** PASS: test_cases\q5\corner_tiny_corner.test
***     pacman layout:          tinyCorner
***     solution length:                28

### Question q5: 3/3 ###
```

Question 4.6 (3 points): Corners Problem: Heuristic
Note: Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.
Implement a non-trivial, consistent heuristic for the CornersProblem in cornersHeuristic.
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Note: AStarCornersAgent is a shortcut for
-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be *admissible*, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be *consistent*, it must additionally hold that if an action has cost c, then taking that action can only cause a drop in heuristic of at most

c.
Remember that admissibility isn't enough to guarantee correctness in graph search – you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in in f-value. Moreover, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the
latter will timeout the autograder. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time
limit).
Grading: Your heuristic must be a non-trivial non-negative consistent heuristic to receive any points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll be graded:

Number of nodes expanded	**Grade**
More than 2000	0/3
At most 2000	1/3
At most 1600	2/3
At most 1200	3/3

Remember: If your heuristic is inconsistent, you will receive *no* credit, so be careful!

```
Question q6
===========
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West', 'West', 'West', 'W
est', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West'
, 'West', 'West', 'South', 'South', 'East', 'East', 'East', 'East', 'South', 'South', 'South
', 'South', 'South', 'South', 'West', 'West', 'South', 'South', 'South', 'West', 'West', 'Ea
st', 'East', 'North', 'North', 'North', 'East', 'East', 'East', 'East', 'East', 'East', 'Eas
t', 'East', 'South', 'South', 'East', 'East', 'East', 'East', 'East', 'North', 'North', 'Eas
t', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'East', 'Eas
t', 'South', 'South', 'South', 'South', 'East', 'East', 'North', 'North', 'East', 'East', 'S
outh', 'South', 'South', 'South', 'South', 'North', 'North', 'North', 'North', 'North', 'Nor
th', 'North', 'West', 'West', 'North', 'North', 'East', 'East', 'North', 'North']
path length: 106
*** PASS: Heuristic resulted in expansion of 901 nodes

### Question q6: 3/3 ###
```

Question 4.7 (4 points): Eating All The Dots

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition which formalizes the food-clearing problem: FoodSearchProblem in searchAgents.py (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For the present project, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next project.) If you have written your general search methods correctly, A* with a null heuristic (equivalent
to uniform-cost search) should quickly find an optimal solution to testSearch with no code change on your part (total cost of 7).

python pacman.py -l testSearch -p AStarFoodSearchAgent

Note: AStarFoodSearchAgent is a shortcut for

-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic

You should find that UCS starts to slow down even for the seemingly simple tinySearch. As a reference, our implementation takes 2.5 seconds to find a path of length 27 after expanding 5057 search
nodes.

Note: Make sure to complete Question 4 before working on Question 7, because Question 7 builds upon your answer for Question 4.

Fill in foodHeuristic in searchAgents.py with a *consistent* heuristic for the FoodSearchProblem.

Try your agent on the trickySearch board:

python pacman.py -l trickySearch -p AStarFoodSearchAgent

Our UCS agent finds the optimal solution in about 13 seconds, exploring over 16,000 nodes.

Any non-trivial non-negative consistent heuristic will receive 1 point. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll get additional points:

N**umber of nodes expanded**	**Grade**
More than 15000	1/4
At most 15000	2/4
At most 12000	3/4
At most 9000	4/4 (full credit; medium)
At most 7000	5/4 (optional extra credit; hard)

Remember: If your heuristic is inconsistent, you will receive *no* credit, so be careful! Can you solve mediumSearch in a short time? If so, we're either very, very impressed, or your heuristic is inconsistent.

```
Question q7
===========
*** PASS: test_cases\q7\food_heuristic_1.test
*** PASS: test_cases\q7\food_heuristic_10.test
*** PASS: test_cases\q7\food_heuristic_11.test
*** PASS: test_cases\q7\food_heuristic_12.test
*** PASS: test_cases\q7\food_heuristic_13.test
*** PASS: test_cases\q7\food_heuristic_14.test
*** PASS: test_cases\q7\food_heuristic_15.test
*** PASS: test_cases\q7\food_heuristic_16.test
*** PASS: test_cases\q7\food_heuristic_17.test
*** PASS: test_cases\q7\food_heuristic_2.test
*** PASS: test_cases\q7\food_heuristic_3.test
*** PASS: test_cases\q7\food_heuristic_4.test
*** PASS: test_cases\q7\food_heuristic_5.test
*** PASS: test_cases\q7\food_heuristic_6.test
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** FAIL: test_cases\q7\food_heuristic_grade_tricky.test
***     expanded nodes: 9551
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 3/4 ###
```

Question 4.8 (3 points): Suboptimal Search
Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. In these cases, we'd still like to find a reasonably good path, quickly. In this section, you'll write an agent that always greedily eats the closest dot. ClosestDotSearchAgent is implemented for you in searchAgents.py, but it's missing a key function that finds a path to the closest dot.
Implement the function findPathToClosestDot in searchAgents.py. Our agent solves this maze (suboptimally!) in under a second with a path cost of 350: python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

Hint: The quickest way to complete findPathToClosestDot is to fill in the AnyFoodSearchProblem, which is missing its goal test. Then, solve that problem with an appropriate search function. The solution should be very short!

Your ClosestDotSearchAgent won't always find the shortest possible path through the maze. Make sure you understand why and try to come up with a small example where repeatedly going to the closest dot does not result in finding the shortest path for eating all the dots.

```
Question q8
===========
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_1.test
***     pacman layout:          Test 1
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_10.test
***     pacman layout:          Test 10
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_11.test
***     pacman layout:          Test 11
***     solution length:                2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_12.test
***     pacman layout:          Test 12
***     solution length:                3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_13.test
***     pacman layout:          Test 13
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_2.test
***     pacman layout:          Test 2
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_3.test
***     pacman layout:          Test 3
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_4.test
***     pacman layout:          Test 4
***     solution length:                3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_5.test
***     pacman layout:          Test 5
***     solution length:                1
```

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_6.test
***     pacman layout:          Test 6
***     solution length:                2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_7.test
***     pacman layout:          Test 7
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_8.test
***     pacman layout:          Test 8
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_9.test
***     pacman layout:          Test 9
***     solution length:                1

### Question q8: 3/3 ###
```

Submission and Evaluation:

This assignment includes an autograder for you to grade your answers on your machine. This can be run with the command:

python autograder.py

```
Finished at 23:12:07

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 3/3
Question q7: 3/4
Question q8: 3/3
------------------
Total: 24/25

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

CODE

SEARCH

```
# search.py
# ---------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.
#
# Attribution Information: The Pacman AI projects were developed at UC Berkeley.
# The core projects and autograders were primarily created by John DeNero
# (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
# Student side autograding was added by Brad Miller, Nick Hay, and
# Pieter Abbeel (pabbeel@cs.berkeley.edu).


"""
In search.py, you will implement generic search algorithms which are called by
Pacman agents (in searchAgents.py).
"""

import util

class SearchProblem:
    """
    This class outlines the structure of a search problem, but doesn't implement
    any of the methods (in object-oriented terminology: an abstract class).

    You do not need to change anything in this class, ever.
    """

    def getStartState(self):
        """
        Returns the start state for the search problem.
        """
        util.raiseNotDefined()

    def isGoalState(self, state):
        """
          state: Search state

        Returns True if and only if the state is a valid goal state.
        """
        util.raiseNotDefined()

    def getSuccessors(self, state):
        """
          state: Search state

        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.
        """
        util.raiseNotDefined()

    def getCostOfActions(self, actions):
        """
         actions: A list of actions to take

        This method returns the total cost of a particular sequence of actions.
        The sequence must be composed of legal moves.
        """
        util.raiseNotDefined()


def tinyMazeSearch(problem):
    """
    Returns a sequence of moves that solves tinyMaze.  For any other maze, the
    sequence of moves will be incorrect, so only use this for tinyMaze.
    """
    from game import Directions
    s = Directions.SOUTH
    w = Directions.WEST
    return  [s, s, w, s, w, w, s, w]

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 ***"
    stack = util.Stack()
    visited = set()

    start_state = problem.getStartState()
    stack.push((start_state, []))

    while not stack.isEmpty():
        current_state, actions = stack.pop()

        if problem.isGoalState(current_state):
            return actions

        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for next_state, action, _ in successors:
                if next_state not in visited:
```

```python
                    stack.push((next_state, actions + [action]))

    return []
    util.raiseNotDefined()

def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    queue = util.Queue()
    visited = set()

    start_state = problem.getStartState()
    queue.push((start_state, []))

    while not queue.isEmpty():
        current_state, actions = queue.pop()

        if problem.isGoalState(current_state):
            return actions

        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for next_state, action, _ in successors:
                if next_state not in visited:
                    queue.push((next_state, actions + [action]))

    return []
    util.raiseNotDefined()

def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    priority_queue = util.PriorityQueue()
    visited = set()

    start_state = problem.getStartState()
    priority_queue.push((start_state, [], 0), 0)

    while not priority_queue.isEmpty():
        current_state, actions, cost = priority_queue.pop()

        if problem.isGoalState(current_state):
            return actions

        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for next_state, action, step_cost in successors:
                if next_state not in visited:
                    priority_queue.push(
                        (next_state, actions + [action], cost + step_cost),
                        cost + step_cost
                    )

    return []
```

```python
    util.raiseNotDefined()

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 ***"
    priority_queue = util.PriorityQueue()
    visited = set()

    start_state = problem.getStartState()
    priority_queue.push((start_state, [], 0), 0)

    while not priority_queue.isEmpty():
        current_state, actions, cost = priority_queue.pop()

        if problem.isGoalState(current_state):
            return actions

        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for next_state, action, step_cost in successors:
                if next_state not in visited:
                    priority_queue.push(
                        (next_state, actions + [action], cost + step_cost),
                        cost + step_cost + heuristic(next_state, problem)
                    )

    return []
    util.raiseNotDefined()


# Abbreviations
bfs = breadthFirstSearch
dfs = depthFirstSearch
astar = aStarSearch
ucs = uniformCostSearch
```

CODE

SEARCH ENGINE

```
# searchAgents.py
# ---------------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.
#
# Attribution Information: The Pacman AI projects were developed at UC Berkeley.
# The core projects and autograders were primarily created by John DeNero
# (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
# Student side autograding was added by Brad Miller, Nick Hay, and
# Pieter Abbeel (pabbeel@cs.berkeley.edu).


"""
This file contains all of the agents that can be selected to control Pacman.  To
select an agent, use the '-p' option when running pacman.py.  Arguments can be
passed to your agent using '-a'.  For example, to load a SearchAgent that uses
depth first search (dfs), run the following command:

> python pacman.py -p SearchAgent -a fn=depthFirstSearch

Commands to invoke other search strategies can be found in the project
description.

Please only change the parts of the file you are asked to.  Look for the lines
that say

"*** YOUR CODE HERE ***"

The parts you fill in start about 3/4 of the way down.  Follow the project
description for details.

Good luck and happy searching!
"""

from game import Directions
from game import Agent
from game import Actions
import util
import time
import search

class GoWestAgent(Agent):
    "An agent that goes West until it can't."

    def getAction(self, state):
        "The agent receives a GameState (defined in pacman.py)."
```

```
        if Directions.WEST in state.getLegalPacmanActions():
            return Directions.WEST
        else:
            return Directions.STOP

#######################################################
# This portion is written for you, but will only work #
#       after you fill in parts of search.py          #
#######################################################

class SearchAgent(Agent):
    """
    This very general search agent finds a path using a supplied search
    algorithm for a supplied search problem, then returns actions to follow that
    path.

    As a default, this agent runs DFS on a PositionSearchProblem to find
    location (1,1)

    Options for fn include:
      depthFirstSearch or dfs
      breadthFirstSearch or bfs


    Note: You should NOT change any code in SearchAgent
    """

    def __init__(self, fn='depthFirstSearch', prob='PositionSearchProblem',
heuristic='nullHeuristic'):
        # Warning: some advanced Python magic is employed below to find the right
functions and problems

        # Get the search function from the name and heuristic
        if fn not in dir(search):
            raise AttributeError(fn + ' is not a search function in search.py.')
        func = getattr(search, fn)
        if 'heuristic' not in func.__code__.co_varnames:
            print('[SearchAgent] using function ' + fn)
            self.searchFunction = func
        else:
            if heuristic in globals().keys():
                heur = globals()[heuristic]
            elif heuristic in dir(search):
                heur = getattr(search, heuristic)
            else:
                raise AttributeError(heuristic + ' is not a function in
searchAgents.py or search.py.')
            print('[SearchAgent] using function %s and heuristic %s' % (fn,
heuristic))
            # Note: this bit of Python trickery combines the search algorithm and the
heuristic
            self.searchFunction = lambda x: func(x, heuristic=heur)

        # Get the search problem type from the name
        if prob not in globals().keys() or not prob.endswith('Problem'):
            raise AttributeError(prob + ' is not a search problem type in
```

```
SearchAgents.py.')
        self.searchType = globals()[prob]
        print('[SearchAgent] using problem type ' + prob)

    def registerInitialState(self, state):
        """
        This is the first time that the agent sees the layout of the game
        board. Here, we choose a path to the goal. In this phase, the agent
        should compute the path to the goal and store it in a local variable.
        All of the work is done in this method!

        state: a GameState object (pacman.py)
        """
        if self.searchFunction == None: raise Exception("No search function provided
for SearchAgent")
        starttime = time.time()
        problem = self.searchType(state) # Makes a new search problem
        self.actions  = self.searchFunction(problem) # Find a path
        totalCost = problem.getCostOfActions(self.actions)
        print('Path found with total cost of %d in %.1f seconds' % (totalCost,
time.time() - starttime))
        if '_expanded' in dir(problem): print('Search nodes expanded: %d' %
problem._expanded)

    def getAction(self, state):
        """
        Returns the next action in the path chosen earlier (in
        registerInitialState).  Return Directions.STOP if there is no further
        action to take.

        state: a GameState object (pacman.py)
        """
        if 'actionIndex' not in dir(self): self.actionIndex = 0
        i = self.actionIndex
        self.actionIndex += 1
        if i < len(self.actions):
            return self.actions[i]
        else:
            return Directions.STOP

class PositionSearchProblem(search.SearchProblem):
    """
    A search problem defines the state space, start state, goal test, successor
    function and cost function.  This search problem can be used to find paths
    to a particular point on the pacman board.

    The state space consists of (x,y) positions in a pacman game.

    Note: this search problem is fully specified; you should NOT change it.
    """

    def __init__(self, gameState, costFn = lambda x: 1, goal=(1,1), start=None,
warn=True, visualize=True):
        """
        Stores the start and goal.
```

```
        gameState: A GameState object (pacman.py)
        costFn: A function from a search state (tuple) to a non-negative number
        goal: A position in the gameState
        """
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        if start != None: self.startState = start
        self.goal = goal
        self.costFn = costFn
        self.visualize = visualize
        if warn and (gameState.getNumFood() != 1 or not gameState.hasFood(*goal)):
            print('Warning: this does not look like a regular search maze')

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

    def getStartState(self):
        return self.startState

    def isGoalState(self, state):
        isGoal = state == self.goal

        # For display purposes only
        if isGoal and self.visualize:
            self._visitedlist.append(state)
            import __main__
            if '_display' in dir(__main__):
                if 'drawExpandedCells' in dir(__main__._display): #@UndefinedVariable
                    __main__._display.drawExpandedCells(self._visitedlist)
#@UndefinedVariable

        return isGoal

    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]:
            x,y = state
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                nextState = (nextx, nexty)
                cost = self.costFn(nextState)
                successors.append( ( nextState, action, cost) )
```

```
        # Bookkeeping for display purposes
        self._expanded += 1 # DO NOT CHANGE
        if state not in self._visited:
            self._visited[state] = True
            self._visitedlist.append(state)

        return successors

    def getCostOfActions(self, actions):
        """
        Returns the cost of a particular sequence of actions. If those actions
        include an illegal move, return 999999.
        """
        if actions == None: return 999999
        x,y= self.getStartState()
        cost = 0
        for action in actions:
            # Check figure out the next state and see whether its' legal
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]: return 999999
            cost += self.costFn((x,y))
        return cost

class StayEastSearchAgent(SearchAgent):
    """
    An agent for position search with a cost function that penalizes being in
    positions on the West side of the board.

    The cost function for stepping into a position (x,y) is 1/2^x.
    """
    def __init__(self):
        self.searchFunction = search.uniformCostSearch
        costFn = lambda pos: .5 ** pos[0]
        self.searchType = lambda state: PositionSearchProblem(state, costFn, (1, 1),
None, False)

class StayWestSearchAgent(SearchAgent):
    """
    An agent for position search with a cost function that penalizes being in
    positions on the East side of the board.

    The cost function for stepping into a position (x,y) is 2^x.
    """
    def __init__(self):
        self.searchFunction = search.uniformCostSearch
        costFn = lambda pos: 2 ** pos[0]
        self.searchType = lambda state: PositionSearchProblem(state, costFn)

def manhattanHeuristic(position, problem, info={}):
    "The Manhattan distance heuristic for a PositionSearchProblem"
    xy1 = position
    xy2 = problem.goal
    return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])

def euclideanHeuristic(position, problem, info={}):
```

```
    "The Euclidean distance heuristic for a PositionSearchProblem"
    xy1 = position
    xy2 = problem.goal
    return ( (xy1[0] - xy2[0]) ** 2 + (xy1[1] - xy2[1]) ** 2 ) ** 0.5

#####################################################
# This portion is incomplete.  Time to write code!  #
#####################################################
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 # 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 ***"

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        return (self.startingPosition, ())  # Empty tuple for indicating no corners
visited initially

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        corners_visited = state[1]
        return len(corners_visited) == len(self.corners)

    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]

            x, y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]

            if not hitsWall:
                # Check if the next position is a corner
                next_position = (nextx, nexty)
                corners_visited = state[1]
                if next_position in self.corners and next_position not in
corners_visited:
                    new_corners_visited = tuple(list(corners_visited) +
[next_position])
                else:
                    new_corners_visited = corners_visited

                successors.append(((next_position, new_corners_visited), action, 1))

        self._expanded += 1 # DO NOT CHANGE
        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)


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).
    """
    position, corners_visited = state
    remaining_corners = set(problem.corners) - set(corners_visited)
    total_distance = 0
    current_position = position
    while remaining_corners:
        min_distance = float('inf')
        for corner in remaining_corners:
            distance = util.manhattanDistance(current_position, corner)
            if distance < min_distance:
                min_distance = distance
                closest_corner = corner
        total_distance += min_distance
        current_position = closest_corner
        remaining_corners.remove(closest_corner)
    return total_distance

# Now, you can run the command:
# python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
class AStarCornersAgent(SearchAgent):
    "A SearchAgent for FoodSearchProblem using A* and your foodHeuristic"
    def __init__(self):
        self.searchFunction = lambda prob: search.aStarSearch(prob, cornersHeuristic)
        self.searchType = CornersProblem

class FoodSearchProblem:
    """
    A search problem associated with finding the a path that collects all of the
    food (dots) in a Pacman game.

    A search state in this problem is a tuple ( pacmanPosition, foodGrid ) where
      pacmanPosition: a tuple (x,y) of integers specifying Pacman's position
      foodGrid:       a Grid (see game.py) of either True or False, specifying
remaining food
    """
    def __init__(self, startingGameState):
        self.start = (startingGameState.getPacmanPosition(),
startingGameState.getFood())
        self.walls = startingGameState.getWalls()
        self.startingGameState = startingGameState
        self._expanded = 0 # DO NOT CHANGE
        self.heuristicInfo = {} # A dictionary for the heuristic to store information

    def getStartState(self):
        return self.start

    def isGoalState(self, state):
        return state[1].count() == 0

    def getSuccessors(self, state):
        "Returns successor states, the actions they require, and a cost of 1."
        successors = []
        self._expanded += 1 # DO NOT CHANGE
```

```
        for direction in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:
            x,y = state[0]
            dx, dy = Actions.directionToVector(direction)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                nextFood = state[1].copy()
                nextFood[nextx][nexty] = False
                successors.append( ( ((nextx, nexty), nextFood), direction, 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"""
        x,y= self.getStartState()[0]
        cost = 0
        for action in actions:
            # figure out the next state and see whether it's legal
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]:
                return 999999
            cost += 1
        return cost

class AStarFoodSearchAgent(SearchAgent):
    "A SearchAgent for FoodSearchProblem using A* and your foodHeuristic"
    def __init__(self):
        self.searchFunction = lambda prob: search.aStarSearch(prob, foodHeuristic)
        self.searchType = FoodSearchProblem

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 ***"


    # previous solution 3/4, calculate manhattan distance to every food node
    x1 = state[0][0]
    y1 = state[0][1]
    foodList = foodGrid.asList()
    distToFood = []
    if len(foodList) > 0:
        # calculate the manhattan distance for each piece of food and take the max
       for x2, y2 in foodList:
            # manhattan distance from state to food
           distToFood.append(abs(x1 - x2) + abs(y1 - y2))
       return max(distToFood)
    return 0


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 ***"
        #NOTE: I wasnt sure if I should rely on a dependency for this question,
        #if the search.py is not included then this should be replaced with the
commented code
```

```
        #use the bfs search algorithm to find the next closest food
        path = search.breadthFirstSearch(problem)
        return path

        #copy/pasted from my search.py, breadth first search, incase search.py not
included
        #initialize data strucutures
        #fringe = util.Queue()
        #isVisitedSet = []

        #add the starting location to the queue and mark as visited
        #start = problem.getStartState()
        #fringe.push((start, [], 0))

        #go until the queue is empty or the goal has been found
        #while not fringe.isEmpty():
        #    state, actions, costSoFar = fringe.pop()
#get the next state
        #    if problem.isGoalState(state):
         #       return actions
        #    if state not in isVisitedSet:
#test if the next potential state has been visited and enqueue and mark as visited
        #        for nextState, direction, cost in problem.getSuccessors(state):
        #            fringe.push((nextState, actions + [direction], costSoFar + cost))
        #        isVisitedSet.append(state)
#mark as visited so it wont be searched again
        #return path

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 ***"
         # Check if the current position has food
        return self.food[x][y]
        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))
```