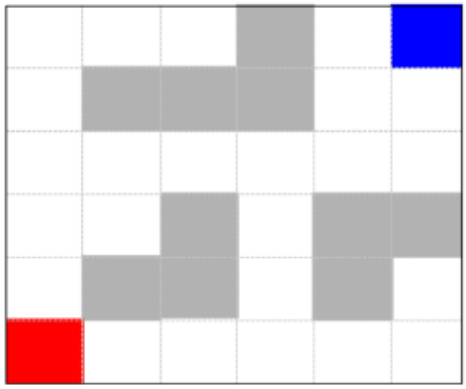
LAB 4/5

PART A : Introductory Problem

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

CODE:

```
from collections import deque
import heapq
import math
class MazeSolver:
    def __init__(self, rows, cols, start, goal, obstacles):
        self.rows = rows
        self.cols = cols
        self.start = start
        self.goal = goal
        self.obstacles = obstacles
        self.explored = set()  # To keep track of explored blocks
        self.search_cost = 0   # To keep track of the total search cost

    def is_valid_move(self, x, y):
        return 1 <= x <= self.rows and 1 <= y <= self.cols and (x, y) not in self.obstacles

    def heuristic(self, x, y):
        # Define a heuristic function (e.g., Manhattan distance)
        return abs(x - self.goal[0]) + abs(y - self.goal[1])

    def solve_maze_dfs(self):
        stack = [(self.start[0], self.start[1])]
        while stack:
            x, y = stack.pop()
            self.search_cost += 1
            if (x, y) == self.goal:
                return True
            if (x, y) not in self.explored:
                self.explored.add((x, y))
                # Push valid neighbors onto the stack
                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    nx, ny = x + dx, y + dy
                    if self.is_valid_move(nx, ny):
                        stack.append((nx, ny))
        return False
```

```
    def solve_maze_bfs(self):
        queue = deque([(self.start[0], self.start[1])])
        while queue:
            x, y = queue.popleft()
            self.search_cost += 1
            if (x, y) == self.goal:
                return True
            if (x, y) not in self.explored:
                self.explored.add((x, y))
                # Enqueue valid neighbors
                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    nx, ny = x + dx, y + dy
                    if self.is_valid_move(nx, ny):
                        queue.append((nx, ny))
        return False

    def solve_maze_dls(self, limit):
        stack = [(self.start[0], self.start[1], 0)]
        while stack:
            x, y, depth = stack.pop()
            self.search_cost += 1
            if depth > limit:
                continue  # Depth limit reached, backtrack
            if (x, y) == self.goal:
                return True
            if (x, y) not in self.explored:
                self.explored.add((x, y))
                # Push valid neighbors onto the stack
                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    nx, ny = x + dx, y + dy
                    if self.is_valid_move(nx, ny):
                        stack.append((nx, ny, depth + 1))
        return False
    def solve_maze_ucs(self):
        # Uniform Cost Search implementation
        pq = [(0, self.start[0], self.start[1])]
        cost_so_far = {(self.start[0], self.start[1]): 0}

        while pq:
            current_cost, x, y = heapq.heappop(pq)
            self.search_cost += 1

            if (x, y) == self.goal:
                return True

            if (x, y) not in self.explored:
                self.explored.add((x, y))

                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    nx, ny = x + dx, y + dy

                    if self.is_valid_move(nx, ny):
                        new_cost = current_cost + 1  # Uniform cost

                        if (nx, ny) not in cost_so_far or new_cost < cost_so_far[(nx, ny)]:
```

```
                        cost_so_far[(nx, ny)] = new_cost
                        heapq.heappush(pq, (new_cost, nx, ny))

        return False

    def solve_maze_gbfs(self):
        # Greedy Best First Search implementation
        pq = [(self.heuristic(self.start[0], self.start[1]), self.start[0], self.start[1])]

        while pq:
            _, x, y = heapq.heappop(pq)
            self.search_cost += 1

            if (x, y) == self.goal:
                return True

            if (x, y) not in self.explored:
                self.explored.add((x, y))

                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    nx, ny = x + dx, y + dy

                    if self.is_valid_move(nx, ny):
                        heapq.heappush(pq, (self.heuristic(nx, ny), nx, ny))

        return False

    def solve_maze_astar(self):
        # A* Algorithm implementation
        pq = [(0, self.start[0], self.start[1])]
        cost_so_far = {(self.start[0], self.start[1]): 0}

        while pq:
            current_cost, x, y = heapq.heappop(pq)
            self.search_cost += 1

            if (x, y) == self.goal:
                return True

            if (x, y) not in self.explored:
                self.explored.add((x, y))

                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    nx, ny = x + dx, y + dy

                    if self.is_valid_move(nx, ny):
                        new_cost = cost_so_far[(x, y)] + 1  # Cost to reach the neighbor
                        if (nx, ny) not in cost_so_far or new_cost < cost_so_far[(nx, ny)]:
                            cost_so_far[(nx, ny)] = new_cost
                            priority = new_cost + self.heuristic(nx, ny)  # A* priority
                            heapq.heappush(pq, (priority, nx, ny))

        return False

    def find_solution(self, search_strategy, depth_limit=None):
```

```
        if search_strategy == 'dfs':
            return self.solve_maze_dfs()
        elif search_strategy == 'bfs':
            return self.solve_maze_bfs()
        elif search_strategy == 'dls':
            if depth_limit is not None:
                return self.solve_maze_dls(depth_limit)
        elif search_strategy == 'ucs':
            return self.solve_maze_ucs()
        elif search_strategy == 'gbfs':
            return self.solve_maze_gbfs()
        elif search_strategy == 'astar':
            return self.solve_maze_astar()
        else:
            return None

if __name__ == '__main__':
    # Increase the recursion limit to a higher value (optional)
    # sys.setrecursionlimit(10000)

    # Read input from the file
    with open('input.txt', 'r') as file:
        rows, cols = map(int, file.readline().split())
        start_x, start_y, goal_x, goal_y = map(int, file.readline().split())
        obstacles = {tuple(map(int, line.split())) for line in file.readline().strip().split()}
        search_strategy = file.readline().strip()
        depth_limit = 3

    solver = MazeSolver(rows, cols, (start_x, start_y), (goal_x, goal_y), obstacles)

    result = solver.find_solution(search_strategy, depth_limit)

    # Write the result to an output file
    with open('output.txt', 'w') as file:
        if result:
            file.write("Sequence of blocks explored:\n")
            for block in sorted(solver.explored):
                file.write(f"{block[0]} {block[1]}\n")
            file.write(f"Total search cost: {solver.search_cost}\n")
        else:
            file.write("No path found.\n")
```

SNIPPETS

C: > Users > dell > Desktop > NITD > 5th sem > AI > Lab 4_and_5 > Lab4_q1.py > MazeSolver

```python
from collections import deque
import heapq
import math
class MazeSolver:
    def __init__(self, rows, cols, start, goal, obstacles):
        self.rows = rows
        self.cols = cols
        self.start = start
        self.goal = goal
        self.obstacles = obstacles
        self.explored = set()  # To keep track of explored blocks
        self.search_cost = 0   # To keep track of the total search cost

    def is_valid_move(self, x, y):
        return 1 <= x <= self.rows and 1 <= y <= self.cols and (x, y) not in self.obstacles

    def heuristic(self, x, y):
        # Define a heuristic function (e.g., Manhattan distance)
        return abs(x - self.goal[0]) + abs(y - self.goal[1])

    def solve_maze_dfs(self):
        stack = [(self.start[0], self.start[1])]
        while stack:
            x, y = stack.pop()
            self.search_cost += 1
            if (x, y) == self.goal:
                return True
            if (x, y) not in self.explored:
                self.explored.add((x, y))
                # Push valid neighbors onto the stack
                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    nx, ny = x + dx, y + dy
```

Lab4_q1.py

C: > Users > dell > Desktop > NITD > 5th sem > AI > Lab 4_and_5 > Lab4_q1.py > MazeSolver

```
                    if self.is_valid_move(nx, ny):
                        stack.append((nx, ny))
        return False

    def solve_maze_bfs(self):
        queue = deque([(self.start[0], self.start[1])])
        while queue:
            x, y = queue.popleft()
            self.search_cost += 1
            if (x, y) == self.goal:
                return True
            if (x, y) not in self.explored:
                self.explored.add((x, y))
                # Enqueue valid neighbors
                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    nx, ny = x + dx, y + dy
                    if self.is_valid_move(nx, ny):
                        queue.append((nx, ny))
        return False

    def solve_maze_dls(self, limit):
        stack = [(self.start[0], self.start[1], 0)]
        while stack:
            x, y, depth = stack.pop()
            self.search_cost += 1
            if depth > limit:
                continue  # Depth limit reached, backtrack
            if (x, y) == self.goal:
                return True
            if (x, y) not in self.explored:
                self.explored.add((x, y))
                # Push valid neighbors onto the stack
```

Lab4_q1.py

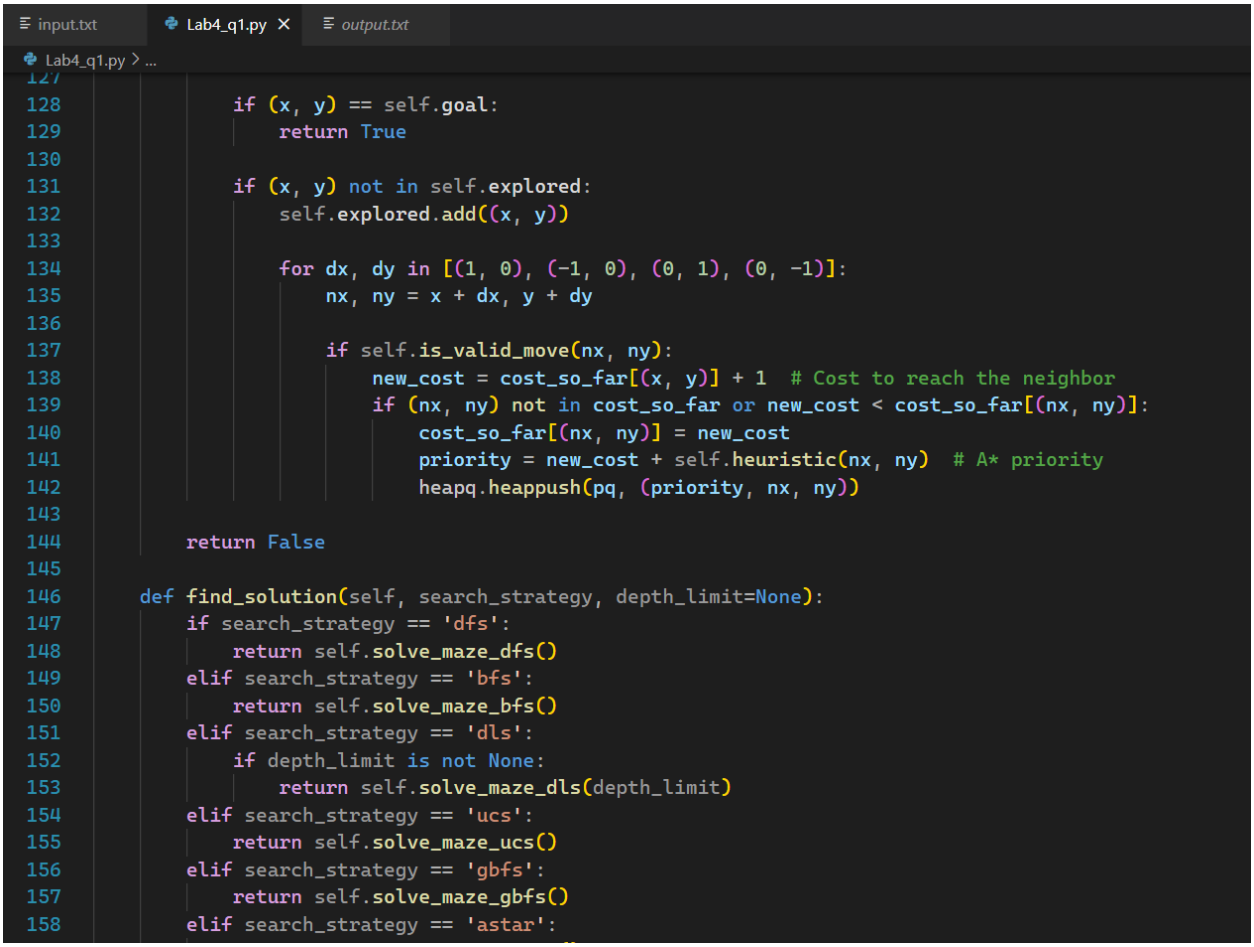
C: > Users > dell > Desktop > NITD > 5th sem > AI > Lab 4_and_5 > Lab4_q1.py > MazeSolver

```python
                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    nx, ny = x + dx, y + dy
                    if self.is_valid_move(nx, ny):
                        stack.append((nx, ny, depth + 1))
        return False
    def solve_maze_ucs(self):
        # Uniform Cost Search implementation
        pq = [(0, self.start[0], self.start[1])]
        cost_so_far = {(self.start[0], self.start[1]): 0}

        while pq:
            current_cost, x, y = heapq.heappop(pq)
            self.search_cost += 1

            if (x, y) == self.goal:
                return True

            if (x, y) not in self.explored:
                self.explored.add((x, y))

                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    nx, ny = x + dx, y + dy

                    if self.is_valid_move(nx, ny):
                        new_cost = current_cost + 1  # Uniform cost

                        if (nx, ny) not in cost_so_far or new_cost < cost_so_far[(nx, ny)]:
                            cost_so_far[(nx, ny)] = new_cost
                            heapq.heappush(pq, (new_cost, nx, ny))

        return False
```

Lab4_q1.py

C: > Users > dell > Desktop > NITD > 5th sem > AI > Lab 4_and_5 > Lab4_q1.py > MazeSolver

```python
    def solve_maze_gbfs(self):
        # Greedy Best First Search implementation
        pq = [(self.heuristic(self.start[0], self.start[1]), self.start[0], self.start[1])]

        while pq:
            _, x, y = heapq.heappop(pq)
            self.search_cost += 1

            if (x, y) == self.goal:
                return True

            if (x, y) not in self.explored:
                self.explored.add((x, y))

                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                    nx, ny = x + dx, y + dy

                    if self.is_valid_move(nx, ny):
                        heapq.heappush(pq, (self.heuristic(nx, ny), nx, ny))

        return False

    def solve_maze_astar(self):
        # A* Algorithm implementation
        pq = [(0, self.start[0], self.start[1])]
        cost_so_far = {(self.start[0], self.start[1]): 0}

        while pq:
            current_cost, x, y = heapq.heappop(pq)
            self.search_cost += 1
```


input.txt
Lab4_q1.py
output.txt
Lab4_q1.py > ...
if (x, y) == self.goal:
return True
if (x, y) not in self.explored:
self.explored.add((x, y))
for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
nx, ny = x + dx, y + dy
if self.is_valid_move(nx, ny):
new_cost = cost_so_far[(x, y)] + 1 # Cost to reach the neighbor
if (nx, ny) not in cost_so_far or new_cost < cost_so_far[(nx, ny)]:
cost_so_far[(nx, ny)] = new_cost
priority = new_cost + self.heuristic(nx, ny) # A* priority
heapq.heappush(pq, (priority, nx, ny))
return False
def find_solution(self, search_strategy, depth_limit=None):
if search_strategy == 'dfs':
return self.solve_maze_dfs()
elif search_strategy == 'bfs':
return self.solve_maze_bfs()
elif search_strategy == 'dls':
if depth_limit is not None:
return self.solve_maze_dls(depth_limit)
elif search_strategy == 'ucs':
return self.solve_maze_ucs()
elif search_strategy == 'gbfs':
return self.solve_maze_gbfs()
elif search_strategy == 'astar':

```python
            return self.solve_maze_astar()
        else:
            return None

if __name__ == '__main__':
    # Increase the recursion limit to a higher value (optional)
    # sys.setrecursionlimit(10000)

    # Read input from the file
    with open('input.txt', 'r') as file:
        rows, cols = map(int, file.readline().split())
        start_x, start_y, goal_x, goal_y = map(int, file.readline().strip().split())
        obstacle_lines = file.readline().strip().split()
        obstacles = {(int(obstacle_lines[i]), int(obstacle_lines[i+1])) for i in range(0, len(obstacle_lines), 2)}
        search_strategy = file.readline().strip()
        depth_limit = 3  # You mentioned depth_limit as 3 in your example

    solver = MazeSolver(rows, cols, (start_x, start_y), (goal_x, goal_y), obstacles)

    result = solver.find_solution(search_strategy, depth_limit)

    # Write the result to an output file
    with open('output.txt', 'w') as file:
        if result:
            file.write("Sequence of blocks explored:\n")
            for block in sorted(solver.explored):
                file.write(f"{block[0]} {block[1]}\n")
            file.write(f"Total search cost: {solver.search_cost}\n")
        else:
            file.write("No path found.\n")
```

INPUT FILE:

input.txt

```
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

```

OUTPUT file


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Code
PS C:\Users\dell\Desktop\NITD\5th sem\AI\Lab 4_and_5> python -u "c:\Users\dell\Desktop\NITD\5th sem\AI\Lab 4_and_5\Lab4_q1.py"
PS C:\Users\dell\Desktop\NITD\5th sem\AI\Lab 4_and_5>

input.txt | Lab4_q1.py | output.txt

output.txt

```
Sequence of blocks explored:
1 1
1 2
1 3
1 4
1 5
1 6
2 2
2 3
2 4
2 6
3 1
3 4
3 6
4 1
4 2
4 3
4 4
5 1
5 4
5 5
5 6
6 4
6 5
Total search cost: 24

```

OBSERVATION:

Best cost (least) is of gbfs and thereafter astart for this particular maze. This might be because of the heuristic used (Manhatten Distance).

no path is found for dls at level=3

(on level=10 the dls returns the cost of 44)

cost of dfs= 47

cost of bfs= 45

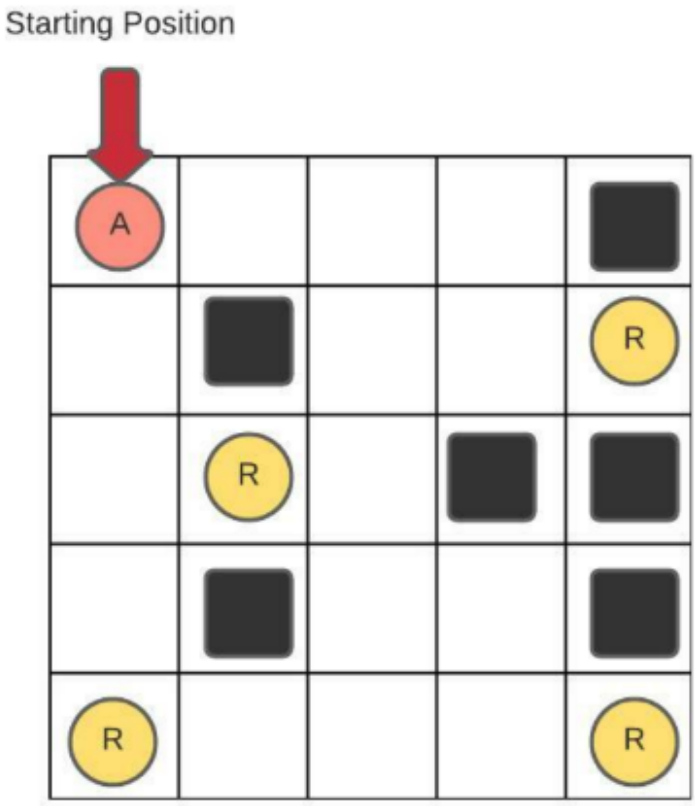
cost of dls= NA

cost of ucs= 24

cost of gbfs= 18

cost of astar= 19

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.

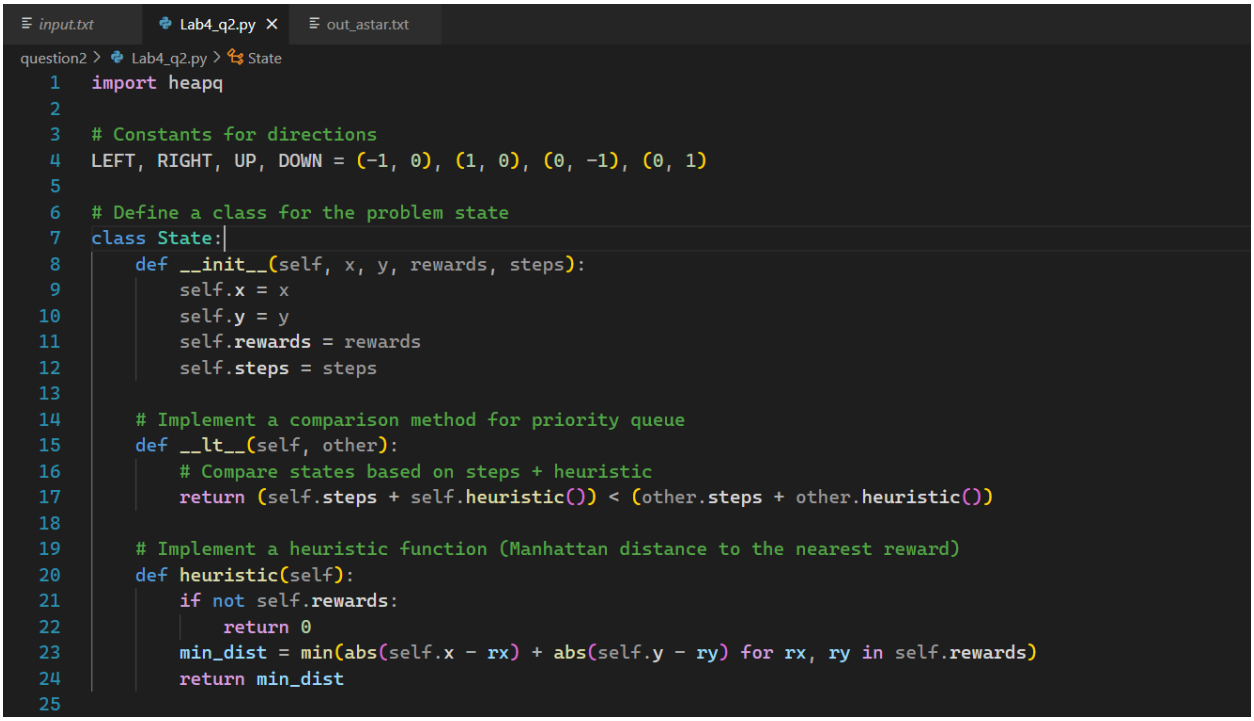
CODE:

```
import heapq

# Constants for directions
LEFT, RIGHT, UP, DOWN = (-1, 0), (1, 0), (0, -1), (0, 1)

# Define a class for the problem state
class State:
    def __init__(self, x, y, rewards, steps):
        self.x = x
        self.y = y
        self.rewards = rewards
        self.steps = steps

    # Implement a comparison method for priority queue
    def __lt__(self, other):
        # Compare states based on steps + heuristic
        return (self.steps + self.heuristic()) < (other.steps + other.heuristic())

    # Implement a heuristic function (Manhattan distance to the nearest reward)
    def heuristic(self):
        if not self.rewards:
            return 0
        min_dist = min(abs(self.x - rx) + abs(self.y - ry) for rx, ry in self.rewards)
        return min_dist

# Implement the A* algorithm
def astar(maze):
    rows, cols = len(maze), len(maze[0])
    start_x, start_y = None, None
    rewards = set()

    # Find the start position and collect rewards
    for x in range(rows):
        for y in range(cols):
            if maze[x][y] == 2:
                start_x, start_y = x, y
            elif maze[x][y] == 3:
                rewards.add((x, y))
```

```
    start_state = State(start_x, start_y, rewards, 0)
    priority_queue = [start_state]
    visited_sequence = []  # Store visited tiles

    while priority_queue:
        current_state = heapq.heappop(priority_queue)
        x, y = current_state.x, current_state.y

        if not current_state.rewards:
            # All rewards collected, exit
            return current_state.steps, visited_sequence

        visited_sequence.append((x, y))  # Record visited tile

        # Explore possible moves (left, right, up, down)
        for dx, dy in [(LEFT), (RIGHT), (UP), (DOWN)]:
            new_x, new_y = x + dx, y + dy

            if 0 <= new_x < rows and 0 <= new_y < cols and maze[new_x][new_y] != 1:
                new_rewards = current_state.rewards.copy()
                if (new_x, new_y) in new_rewards:
                    new_rewards.remove((new_x, new_y))

                new_state = State(new_x, new_y, new_rewards, current_state.steps + 1)
                heapq.heappush(priority_queue, new_state)

    # No path to collect all rewards
    return -1, visited_sequence

# Read input from a text file
def read_maze_from_file(filename):
    with open(filename, 'r') as file:
        maze = [[int(tile) for tile in line.strip().split()] for line in file.readlines()]
    return maze

# Write visited sequence to an output file
def write_visited_sequence_to_file(sequence, filename):
    with open(filename, 'w') as file:
        for x, y in sequence:
            file.write(f"{x} {y}\n")

# Define the input and output filenames
input_filename = 'input.txt'
output_filename = 'out_astar.txt'

# Read the maze from the input file
maze = read_maze_from_file(input_filename)

# Run the A* algorithm
result, visited_sequence = astar(maze)

# Write the visited sequence to the output file
write_visited_sequence_to_file(visited_sequence, output_filename)
```

```
if result == -1:
    print("No path to collect all rewards.")
else:
    print(f"Minimum steps to collect all rewards: {result}")
```

CODE SNIPPETS:

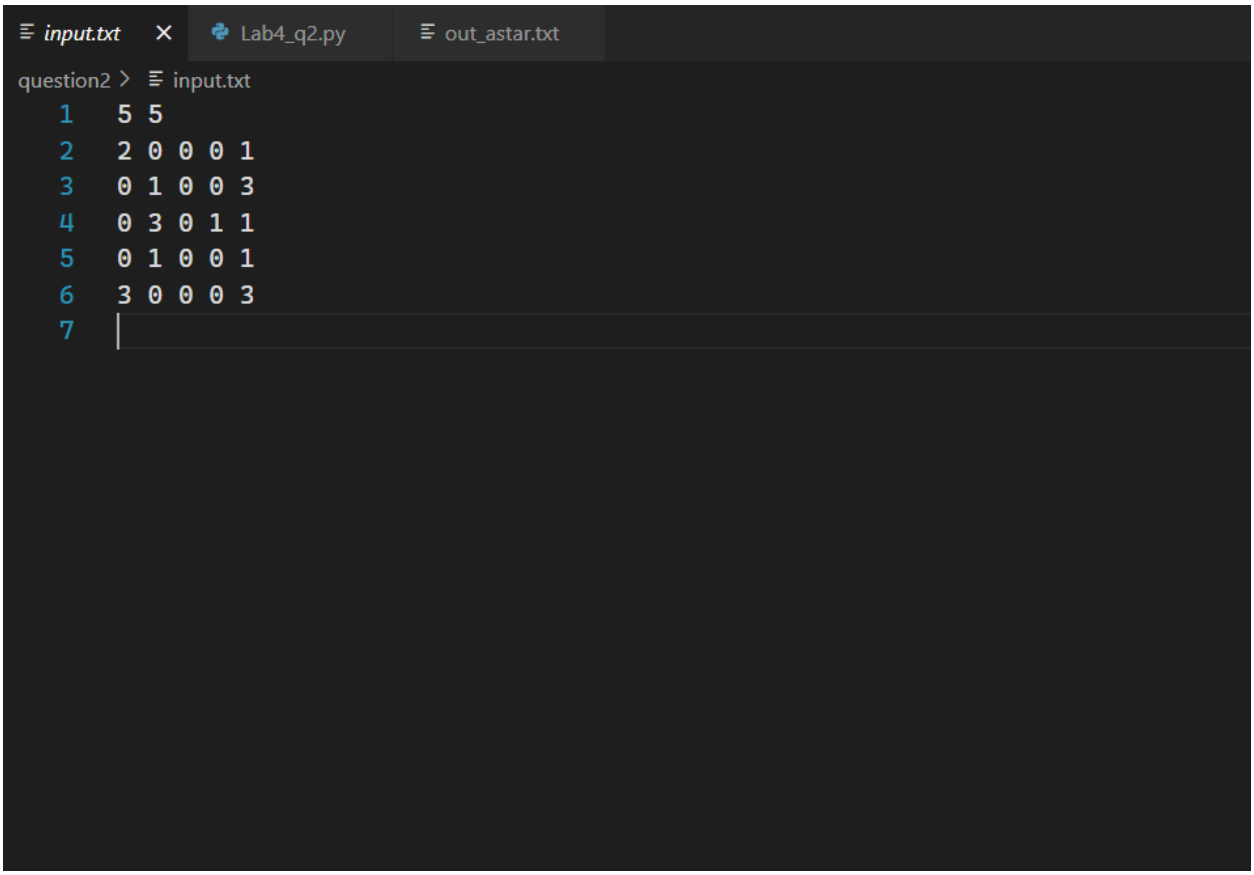


```python
# Implement the A* algorithm
def astar(maze):
    rows, cols = len(maze), len(maze[0])
    start_x, start_y = None, None
    rewards = set()

    # Find the start position and collect rewards
    for x in range(rows):
        for y in range(cols):
            if maze[x][y] == 2:
                start_x, start_y = x, y
            elif maze[x][y] == 3:
                rewards.add((x, y))

    start_state = State(start_x, start_y, rewards, 0)
    priority_queue = [start_state]
    visited_sequence = []  # Store visited tiles

    while priority_queue:
        current_state = heapq.heappop(priority_queue)
        x, y = current_state.x, current_state.y

        if not current_state.rewards:
            # All rewards collected, exit
            return current_state.steps, visited_sequence

        visited_sequence.append((x, y))  # Record visited tile

        # Explore possible moves (left, right, up, down)
        for dx, dy in [(LEFT), (RIGHT), (UP), (DOWN)]:
            new_x, new_y = x + dx, y + dy
```

```
            if 0 <= new_x < rows and 0 <= new_y < cols and maze[new_x][new_y] != 1:
                new_rewards = current_state.rewards.copy()
                if (new_x, new_y) in new_rewards:
                    new_rewards.remove((new_x, new_y))

                new_state = State(new_x, new_y, new_rewards, current_state.steps + 1)
                heapq.heappush(priority_queue, new_state)

    # No path to collect all rewards
    return -1, visited_sequence

# Read input from a text file
def read_maze_from_file(filename):
    with open(filename, 'r') as file:
        maze = [[int(tile) for tile in line.strip().split()] for line in file.readlines()]
    return maze

# Write visited sequence to an output file
def write_visited_sequence_to_file(sequence, filename):
    with open(filename, 'w') as file:
        for x, y in sequence:
            file.write(f"{x} {y}\n")

# Define the input and output filenames
input_filename = 'input.txt'
output_filename = 'out_astar.txt'

# Read the maze from the input file
maze = read_maze_from_file(input_filename)
```

```
# Run the A* algorithm
result, visited_sequence = astar(maze)

# Write the visited sequence to the output file
write_visited_sequence_to_file(visited_sequence, output_filename)

if result == -1:
    print("No path to collect all rewards.")
else:
    print(f"Minimum steps to collect all rewards: {result}")
```

INPUT File:



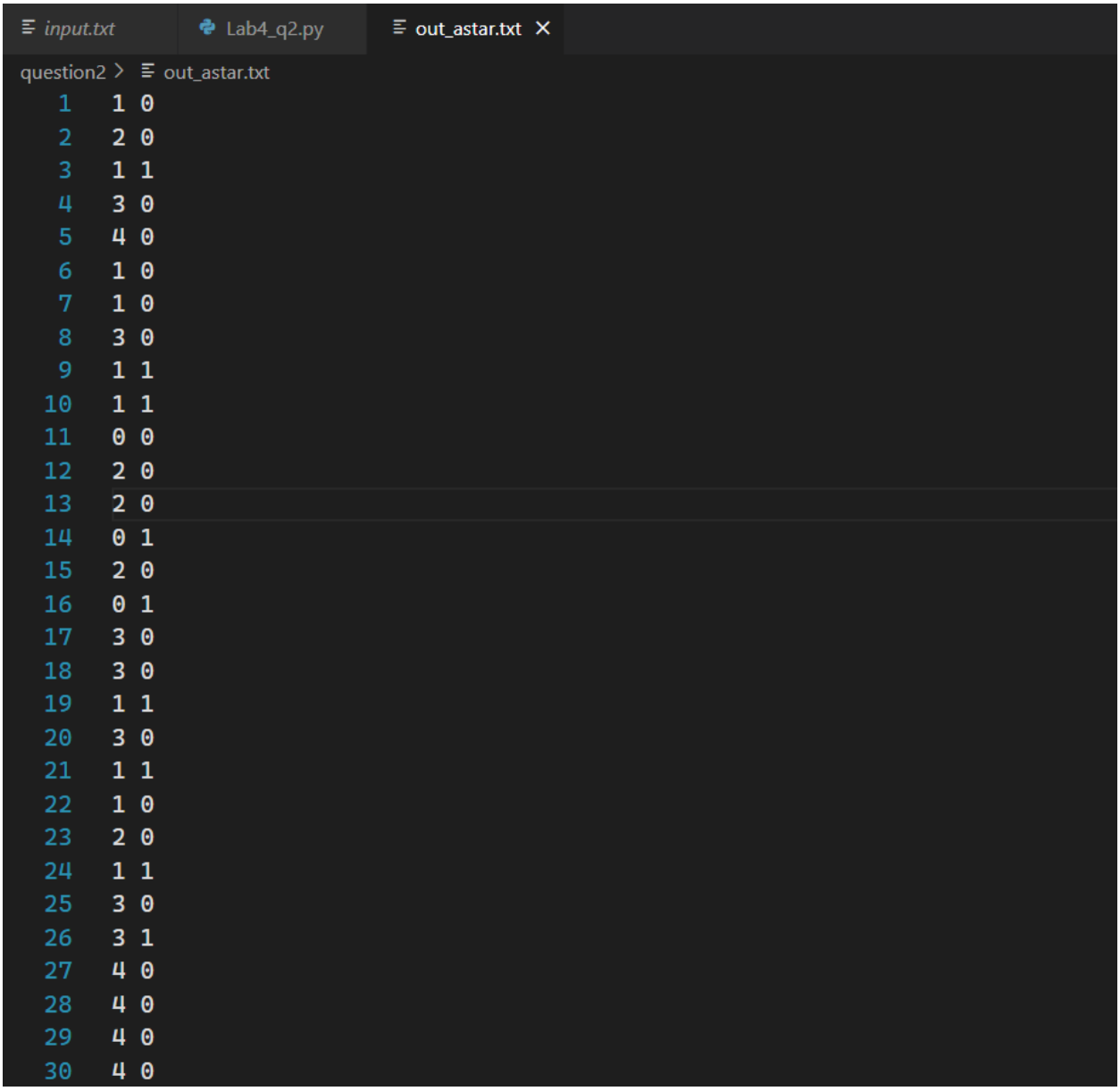
2 for start

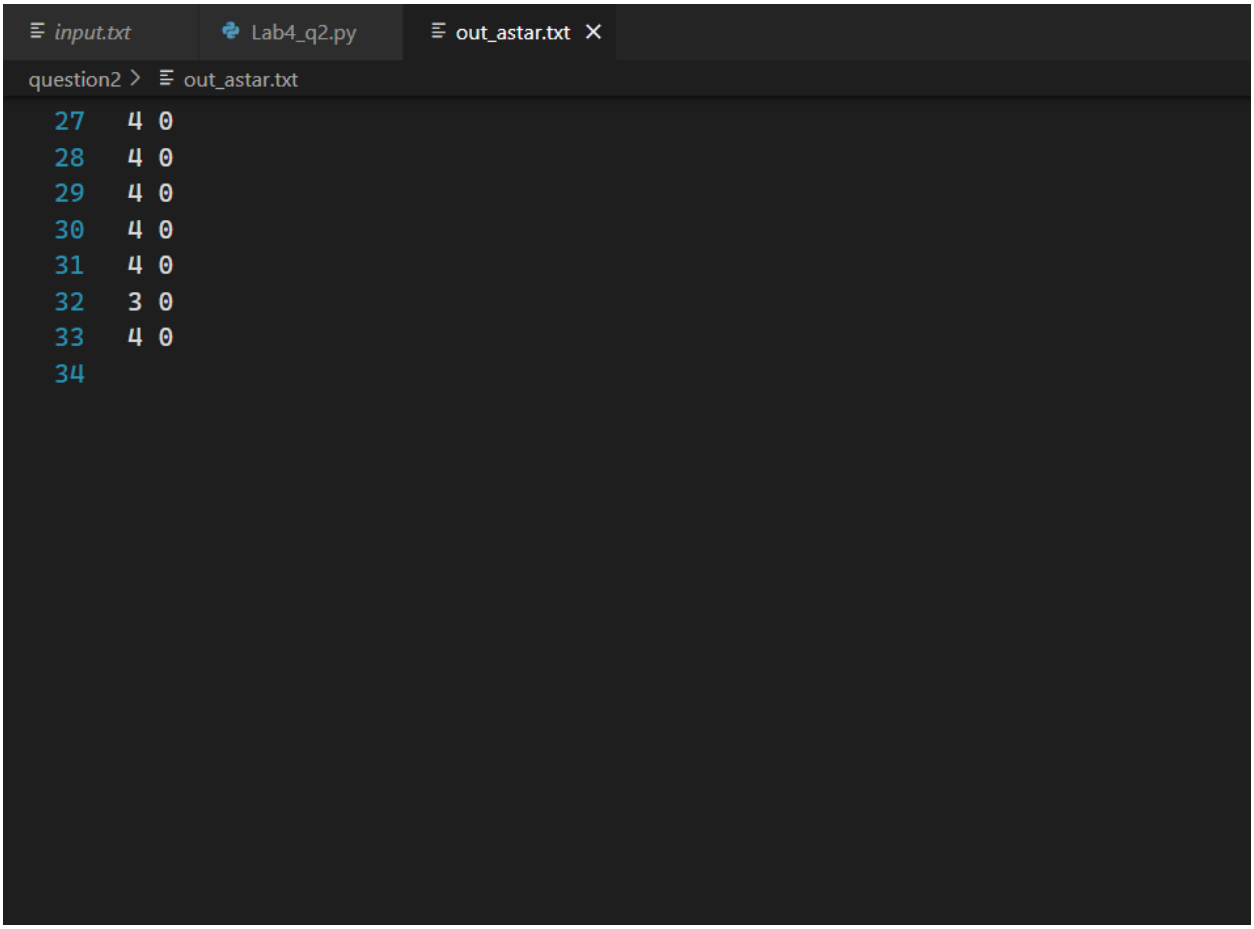
0 for clear path

3 for reward

1 for obstacle

OUTPUT File:


input.txt
Lab4_q2.py
out_astar.txt
question2 > out_astar.txt
1 1 0
2 2 0
3 1 1
4 3 0
5 4 0
6 1 0
7 1 0
8 3 0
9 1 1
10 1 1
11 0 0
12 2 0
13 2 0
14 0 1
15 2 0
16 0 1
17 3 0
18 3 0
19 1 1
20 3 0
21 1 1
22 1 0
23 2 0
24 1 1
25 3 0
26 3 1
27 4 0
28 4 0
29 4 0
30 4 0



OBSERVATION:

It takes 33 steps for the agent to reach all the rewards in the problem maze using astar algorithm

Exploratory Problem

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

CODE:

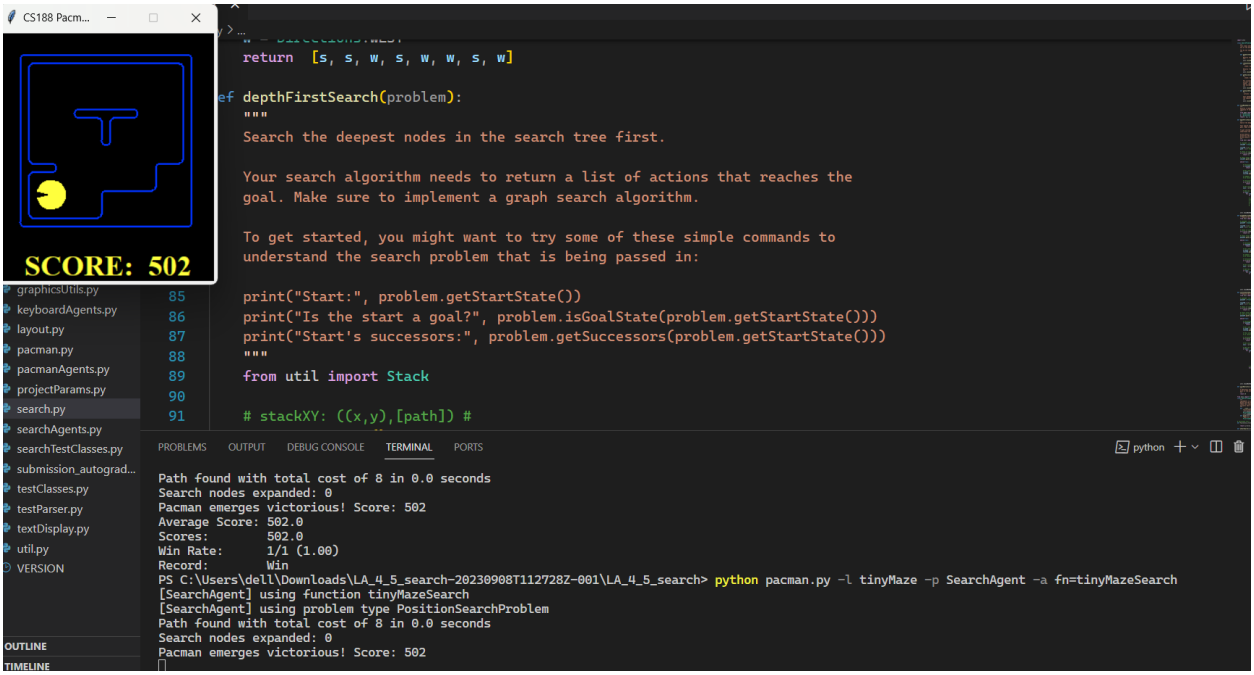
```
75  def depthFirstSearch(problem):
76      """
77      Search the deepest nodes in the search tree first.
78
79      Your search algorithm needs to return a list of actions that reaches the
80      goal. Make sure to implement a graph search algorithm.
81
82      To get started, you might want to try some of these simple commands to
83      understand the search problem that is being passed in:
84
85      print("Start:", problem.getStartState())
86      print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
87      print("Start's successors:", problem.getSuccessors(problem.getStartState()))
88      """
89      from util import Stack
90
91      # stackXY: ((x,y),[path]) #
92      stackXY = Stack()
93
94      visited = [] # Visited states
95      path = [] # Every state keeps it's path from the starting state
96
97      # Check if initial state is goal state #
98      if problem.isGoalState(problem.getStartState()):
99          return []
100
101     # Start from the beginning and find a solution, path is an empty list #
102     stackXY.push((problem.getStartState(),[]))
103
```

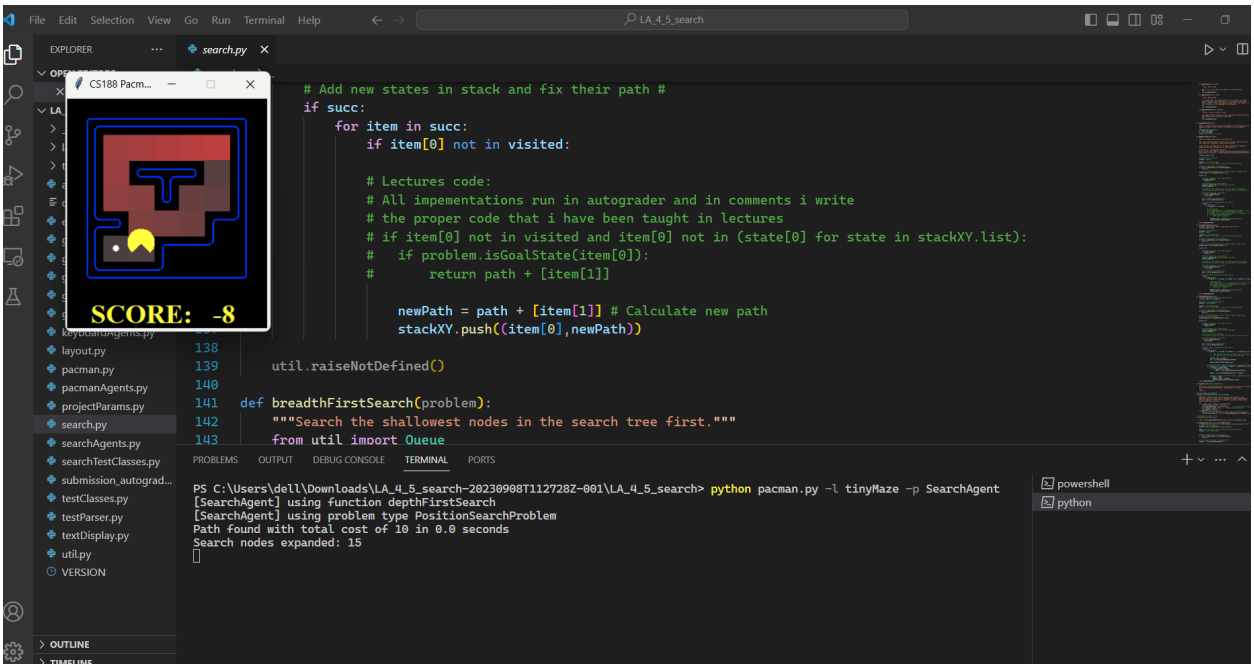
search.py > ...

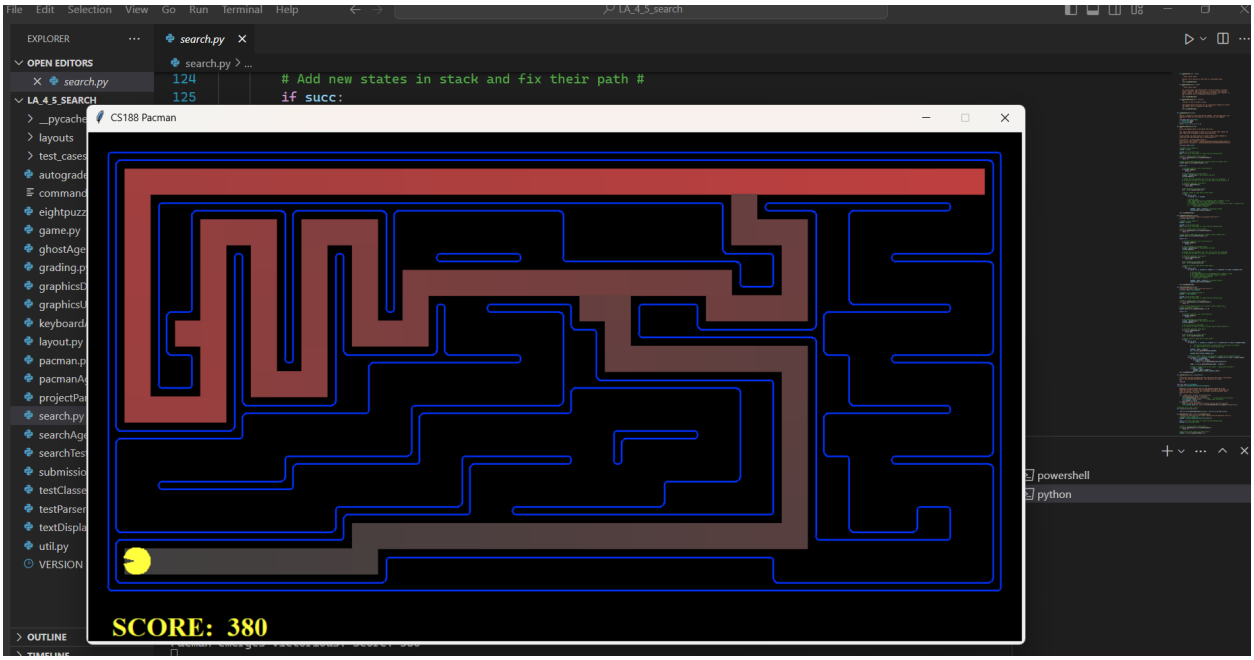
```python
    while(True):

        # Terminate condition: can't find solution #
        if stackXY.isEmpty():
            return []

        # Get informations of current state #
        xy,path = stackXY.pop() # Take position and path
        visited.append(xy)

        # Comment this and uncomment 125. This only works for autograder    #
        # In lectures we check if a state is a goal when we find successors #

        # Terminate condition: reach goal #
        if problem.isGoalState(xy):
            return path

        # Get successors of current state #
        succ = problem.getSuccessors(xy)

        # Add new states in stack and fix their path #
        if succ:
            for item in succ:
                if item[0] not in visited:

                    # Lectures code:
                    # All impementations run in autograder and in comments i write
                    # the proper code that i have been taught in lectures
                    # if item[0] not in visited and item[0] not in (state[0] for state in stackXY.list):
                    #   if problem.isGoalState(item[0]):
                    #       return path + [item[1]]
```

```python
                    #       return path + [item[1]]

                    newPath = path + [item[1]] # Calculate new path
                    stackXY.push((item[0],newPath))

    util.raiseNotDefined()
```

OUTPUT:


CS188 Pacm...
SCORE: 502
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()))
"""
from util import Stack
# stackXY: ((x,y),[path]) #
graphicsUtils.py
keyboardAgents.py
layout.py
pacman.py
pacmanAgents.py
projectParams.py
search.py
searchAgents.py
searchTestClasses.py
submission_autograd...
testClasses.py
testParser.py
textDisplay.py
util.py
VERSION
OUTLINE
TIMELINE
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
python
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 0
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores: 502.0
Win Rate: 1/1 (1.00)
Record: Win
PS C:\Users\dell\Downloads\LA_4_5_search-20230908T112728Z-001\LA_4_5_search> python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
[SearchAgent] using function tinyMazeSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 0
Pacman emerges victorious! Score: 502


File Edit Selection View Go Run Terminal Help
LA_4_5_search
EXPLORER
search.py
CS188 Pacm...
SCORE: -8
# Add new states in stack and fix their path #
if succ:
for item in succ:
if item[0] not in visited:
# Lectures code:
# All impementations run in autograder and in comments i write
# the proper code that i have been taught in lectures
# if item[0] not in visited and item[0] not in (state[0] for state in stackXY.list):
# if problem.isGoalState(item[0]):
# return path + [item[1]]
newPath = path + [item[1]] # Calculate new path
stackXY.push((item[0],newPath))
util.raiseNotDefined()
def breadthFirstSearch(problem):
"""Search the shallowest nodes in the search tree first."""
from util import Queue
layout.py
pacman.py
pacmanAgents.py
projectParams.py
search.py
searchAgents.py
searchTestClasses.py
submission_autograd...
testClasses.py
testParser.py
textDisplay.py
util.py
VERSION
OUTLINE
TIMELINE
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
powershell
python
PS C:\Users\dell\Downloads\LA_4_5_search-20230908T112728Z-001\LA_4_5_search> python pacman.py -l tinyMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15


CS188 Pacman
SCORE: 380
search.py
# Add new states in stack and fix their path #
if succ:
powershell
python


CS188 Pacman
SCORE: -209
search.py
Path found with
Search nodes ex
Pacman emerges
Average Score:
Scores:
Win Rate: 1/1 (1.00)
Record: Win
PS C:\Users\dell\Downloads\LA_4_5_search-20230908T112728Z-001\LA_4_5_search> python pacman.py -l bigMaze -z .5 -p SearchAg
ent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
powershell
python

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.

CODE:

search.py

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    from util import Queue

    # queueXY: ((x,y),[path]) #
    queueXY = Queue()

    visited = [] # Visited states
    path = [] # Every state keeps it's path from the starting state

    # Check if initial state is goal state #
    if problem.isGoalState(problem.getStartState()):
        return []

    # Start from the begin (function) getStartState: Any is empty list #
    queueXY.push((problem.getStartState(),[]))

    while(True):

        # Terminate condition: can't find solution #
        if queueXY.isEmpty():
            return []

        # Get informations of current state #
        xy,path = queueXY.pop() # Take position and path
        visited.append(xy)

        # Comment this and uncomment 179. This is only works for autograder
```

```
        # Comment this and uncomment 179. This is only works for autograder
        # In lectures we check if a state is a goal when we find successors

        # Terminate condition: reach goal #
        if problem.isGoalState(xy):
            return path

        # Get successors of current state #
        succ = problem.getSuccessors(xy)

        # Add new states in queue and fix their path #
        if succ:
            for item in succ:
                if item[0] not in visited and item[0] not in (state[0] for state in queueXY.list):

                    # Lectures code:
                    # All impementations run in autograder and in comments i write
                    # the proper code that i have been taught in lectures
                    # if problem.isGoalState(item[0]):
                    #   return path + [item[1]]

                    newPath = path + [item[1]] # Calculate new path
                    queueXY.push((item[0],newPath))

    util.raiseNotDefined()

```

OUTPUT:

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.

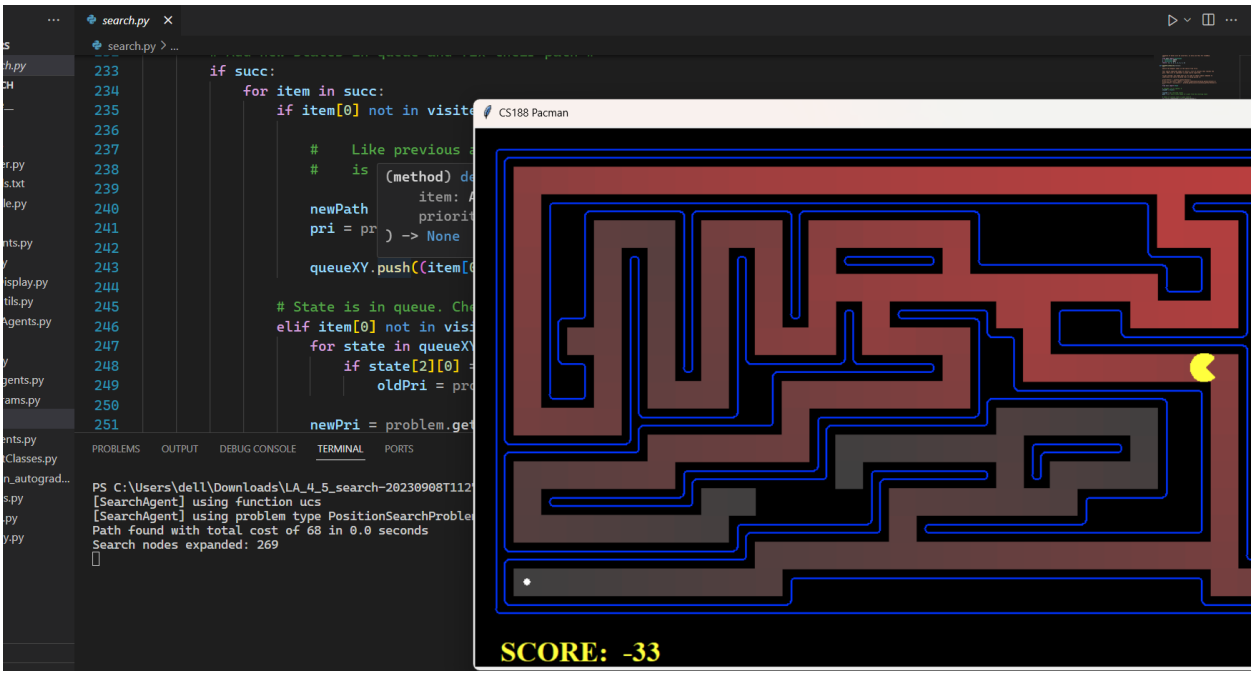
CODE:

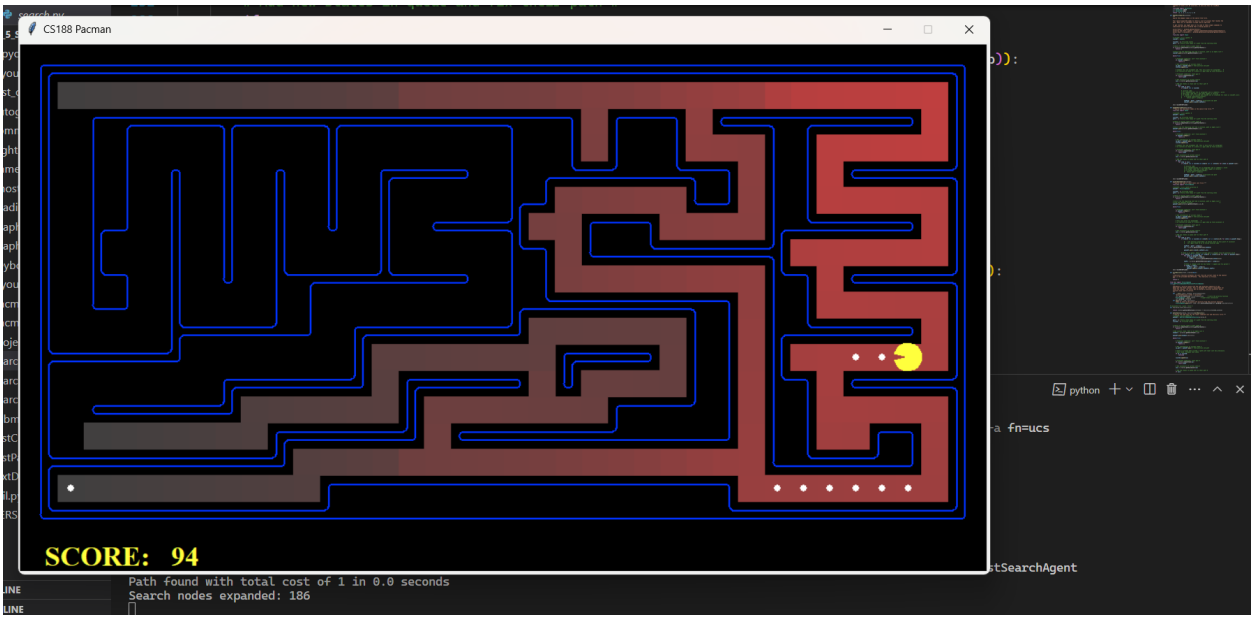
```python
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    from util import PriorityQueue

    # queueXY: ((x,y),[path],priority) #
    queueXY = PriorityQueue()

    visited = [] # Visited states
    path = [] # Every state keeps it's path from the starting state

    # Check if initial state is goal state #
    if problem.isGoalState(problem.getStartState()):
        return []

    # Start from the beginning and find a solution, path is empty list #
    # with the cheapest priority                                       #
    queueXY.push((problem.getStartState(),[]),0)

    while(True):

        # Terminate condition: can't find solution #
        if queueXY.isEmpty():
            return []

        # Get informations of current state #
        xy,path = queueXY.pop() # Take position and path
        visited.append(xy)

        # This only works for autograder    #
        # In lectures we check if a state is a goal when we find successors #
```

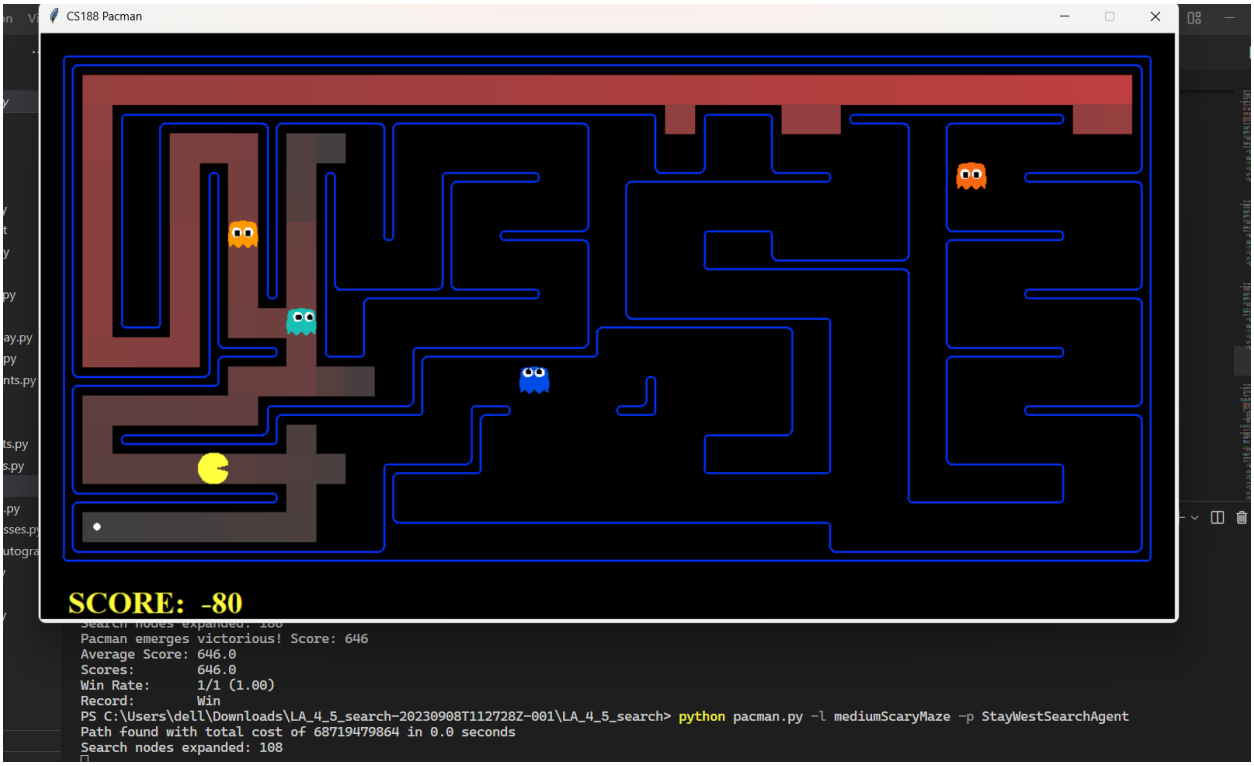
search.py

```python
        # Terminate condition: reach goal #
        if problem.isGoalState(xy):
            return path

        # Get successors of current state #
        succ = problem.getSuccessors(xy)

        # Add new states in queue and fix their path #
        if succ:
            for item in succ:
                if item[0] not in visited and (item[0] not in (state[2][0] for state in queueXY.heap)):

                    #    Like previous algorithms: we should check in this point if successor
                    #    is a goal state so as to follow lectures code

                    newPath = path + [item[1]]
                    pri = problem.getCostOfActions(newPath)

                    queueXY.push((item[0],newPath),pri)

                # State is in queue. Check if current path is cheaper from the previous one #
                elif item[0] not in visited and (item[0] in (state[2][0] for state in queueXY.heap)):
                    for state in queueXY.heap:
                        if state[2][0] == item[0]:
                            oldPri = problem.getCostOfActions(state[2][1])

                    newPri = problem.getCostOfActions(path + [item[1]])

                    # State is cheaper with his hew father -> update and fix parent #
                    if oldPri > newPri:
                        newPath = path + [item[1]]
```

OUTPUT:


search.py
CS188 Pacman
SCORE: -33
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
[SearchAgent] using function ucs
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269


CS188 Pacman
SCORE: 94
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
fn=ucs



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.

CODE:

```python
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

from util import PriorityQueue
class MyPriorityQueueWithFunction(PriorityQueue):
    """
    Implements a priority queue with the same push/pop signature of the
    Queue and the Stack classes. This is designed for drop-in replacement for
    those two classes. The caller has to provide a priority function, which
    extracts each item's priority.
    """
    def  __init__(self, problem, priorityFunction):
        "priorityFunction (item) -> priority"
        self.priorityFunction = priorityFunction      # store the priority function
        PriorityQueue.__init__(self)        # super-class initializer
        self.problem = problem
    def push(self, item, heuristic):
        "Adds an item to the queue with priority from the priority function"
        PriorityQueue.push(self, item, self.priorityFunction(self.problem,item,heuristic))

# Calculate f(n) = g(n) + h(n) #
def f(problem,state,heuristic):

```

```python
    return problem.getCostOfActions(state[1]) + heuristic(state[0],problem)

def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    # queueXY: ((x,y),[path]) #
    queueXY = MyPriorityQueueWithFunction(problem,f)

    path = [] # Every state keeps it's path from the starting state
    visited = [] # Visited states


    # Check if initial stat (parameter) problem: Any
    if problem.isGoalState(problem.getStartState()):
        return []

    # Add initial state. Path is an empty list #
    element = (problem.getStartState(),[])

    queueXY.push(element,heuristic)

    while(True):

        # Terminate condition: can't find solution #
        if queueXY.isEmpty():
            return []
```

```
312         # Get informations of current state #
313         xy,path = queueXY.pop() # Take position and path
314 
315         # State is already been visited. A path with lower cost has previously
316         # been found. Overpass this state
317         if xy in visited:
318             continue
319 
320         visited.append(xy)
321 
322         # Terminate condition: reach goal #
323         if problem.isGoalState(xy):
324             return path
325 
326         # Get successors of current state #
327         succ = problem.getSuccessors(xy)
328 
329         # Add new states in queue and fix their path #
330         if succ:
331             for item in succ:
332                 if item[0] not in visited:
333 
334                     # Like previous algorithms: we should check in this point if successor
335                     # is a goal state so as to follow lectures code
336 
```

```
336 
337                     newPath = path + [item[1]] # Fix new path
338                     element = (item[0],newPath)
339                     queueXY.push(element,heuristic)
340     util.raiseNotDefined()
341 
```

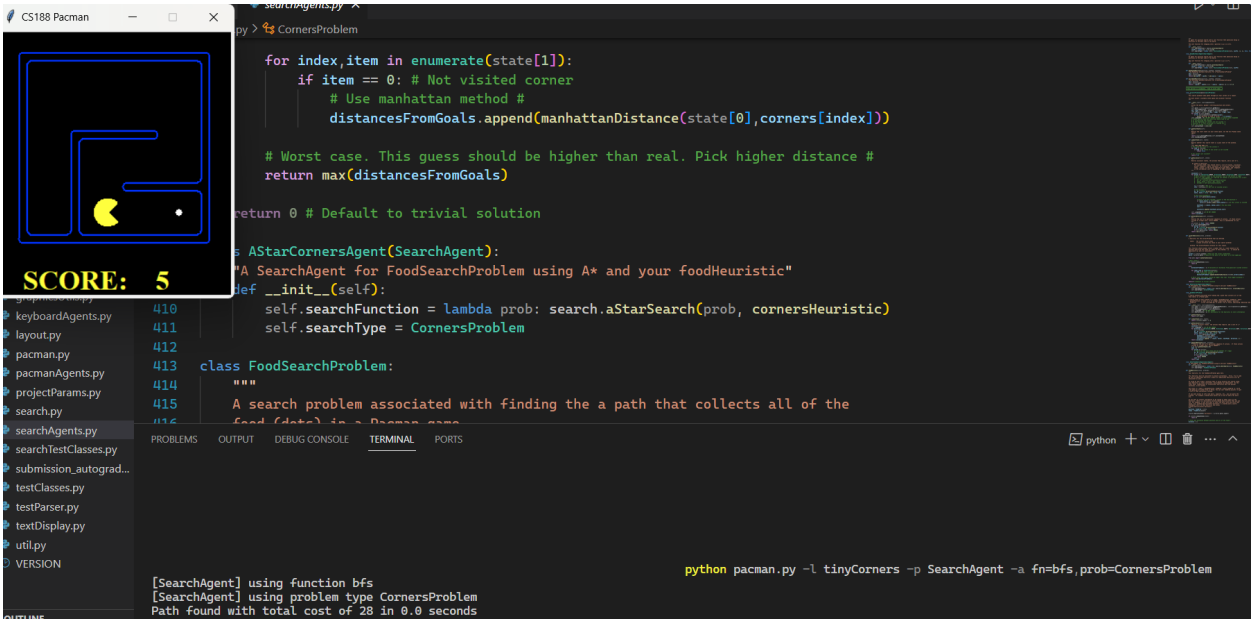
OUTPUT

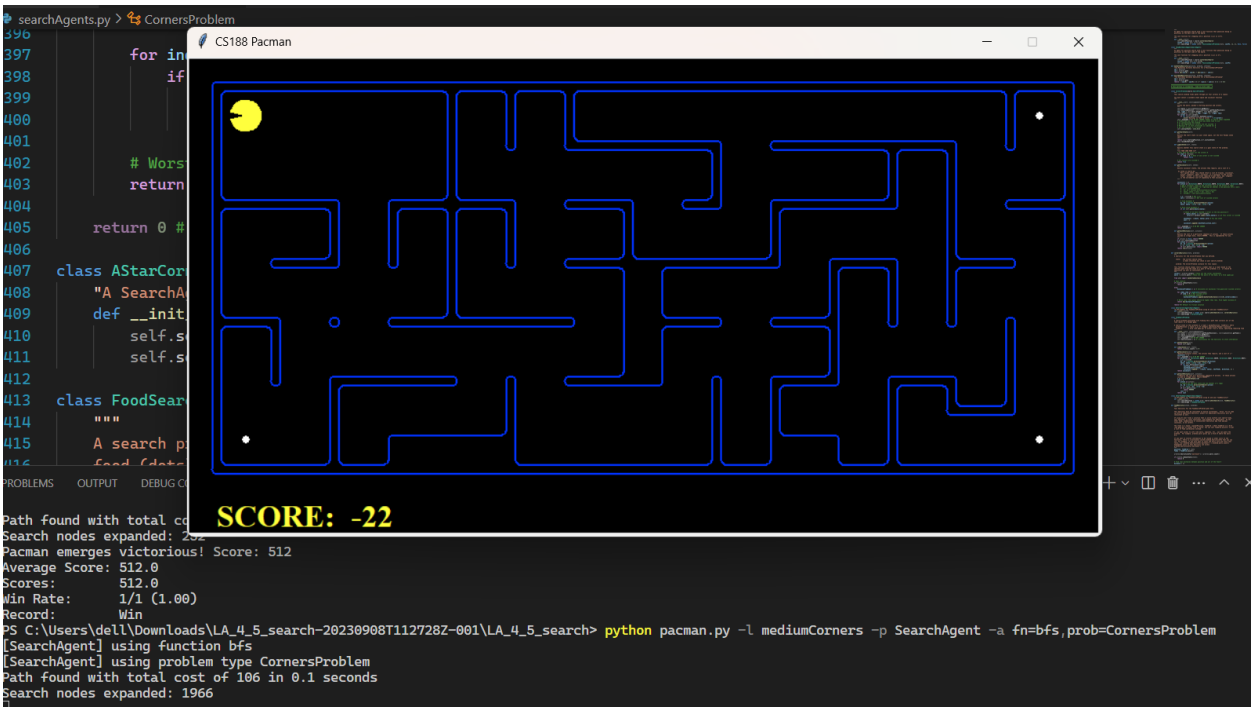
Question 4.5 (3 points): Finding All the Corners
The real power of A* will only be apparent with a more challenging search problem. Now, it's time to
formulate a new problem and design a heuristic for it.
In corner mazes, there are four dots, one in each corner. Our new search problem is to find the 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.

CODE:

```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)

    from util import manhattanDistance

    # Goal state #
    if problem.isGoalState(state):
        return 0

    else:
        distancesFromGoals = [] # Calculate all distances from goals(not visited corners)

        for index,item in enumerate(state[1]):
            if item == 0: # Not visited corner
                # Use manhattan method #
                distancesFromGoals.append(manhattanDistance(state[0],corners[index]))

        # Worst case. This guess should be higher than real. Pick higher distance #
        return max(distancesFromGoals)

    return 0 # Default to trivial solution
```

OUTPUT:



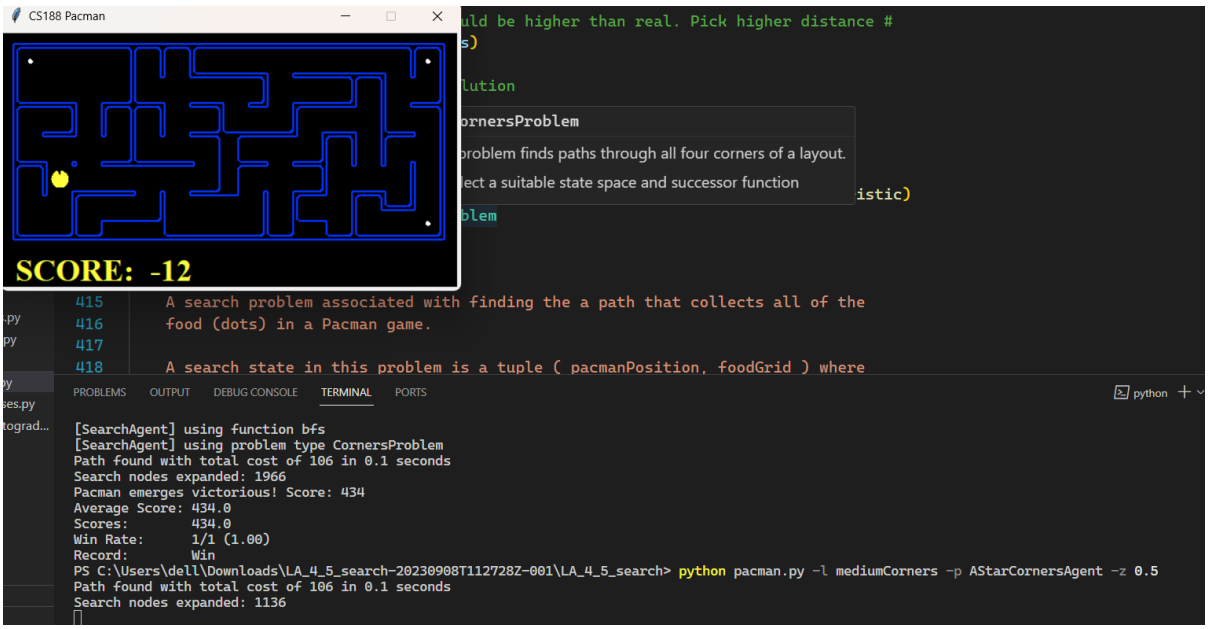


▼ Question 4.6 (3 points): Corners Problem: Heuristic
Note: Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.
Implement a non-trivial, consistent heuristic for the CornersProblem in cornersHeuristic.

CODE:

```
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
```

OUTPUT



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).

CODE:

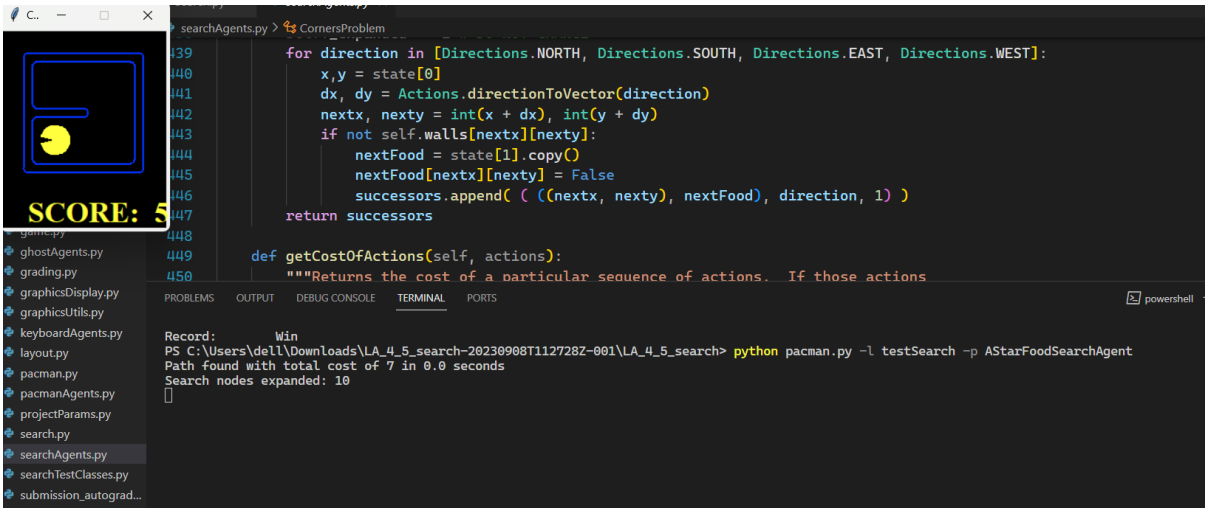
search.py searchAgents.py

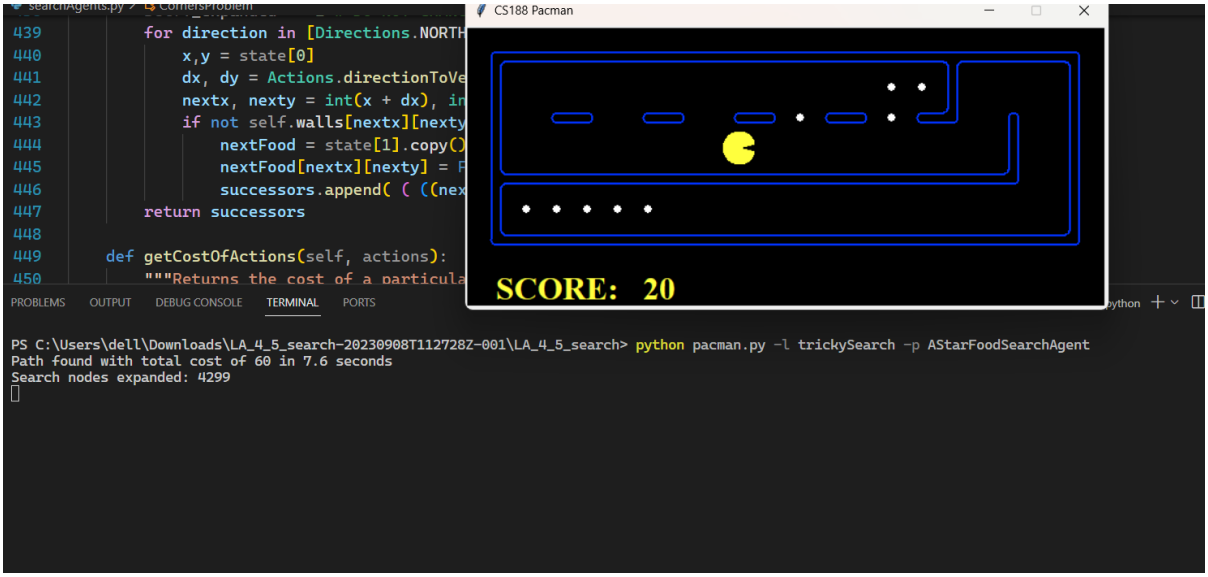
searchAgents.py > CornersProblem

```python
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]
```

```python
            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
```

OUTPUT:





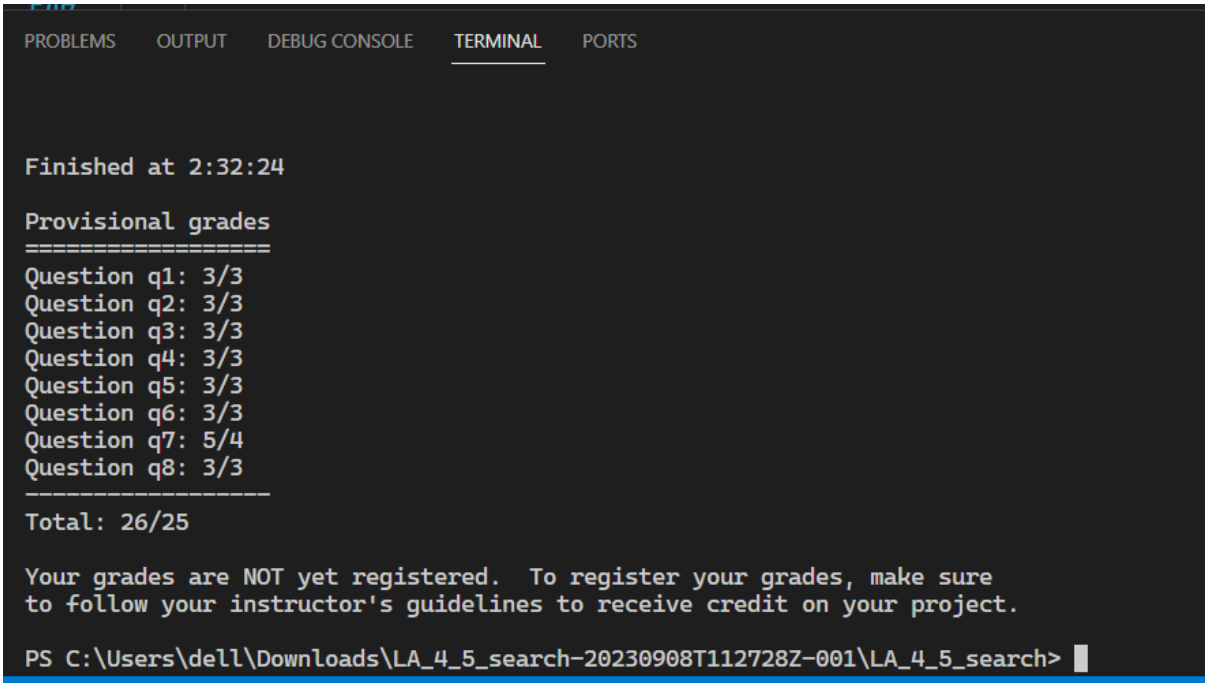
Question 4.8 (3 points): Suboptimal Search
Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard.
In these cases, we'd still like to find a reasonably good path, quickly. In this section, you'll write an
agent that always greedily eats the closest dot.
ClosestDotSearchAgent is implemented for you in searchAgents.py, but it's missing a key function that finds a path to the closest dot.

CODE:

```
537     def findPathToClosestDot(self, gameState):
538         """
539         Returns a path (a list of actions) to the closest dot, starting from
540         gameState.
541         """
542         # Here are some useful elements of the startState
543         startPosition = gameState.getPacmanPosition()
544         food = gameState.getFood()
545         walls = gameState.getWalls()
546         problem = AnyFoodSearchProblem(gameState)
547 
548         temp = food.asList()
549 
550         from search import breadthFirstSearch
551 
552         # Bfs finds closest food first. #
553         return breadthFirstSearch(problem) # Return actions
554         util.raiseNotDefined()
555 
```

OUTPUT

Submission and Evaluation:


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Finished at 2:32:24
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: 5/4
Question q8: 3/3

Total: 26/25
Your grades are NOT yet registered. To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
PS C:\Users\dell\Downloads\LA_4_5_search-20230908T112728Z-001\LA_4_5_search>