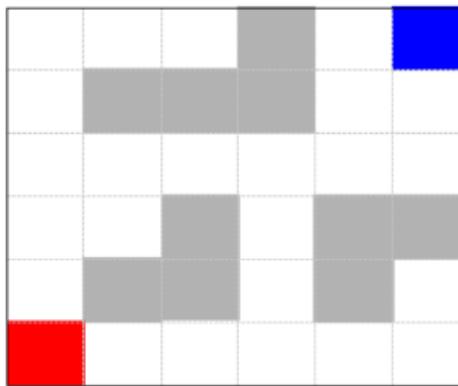


LAB 4/5

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
 1  from collections import deque
 2  import heapq
 3  import math
 4  class MazeSolver:
 5      def __init__(self, rows, cols, start, goal, obstacles):
 6          self.rows = rows
 7          self.cols = cols
 8          self.start = start
 9          self.goal = goal
10          self.obstacles = obstacles
11          self.explored = set() # To keep track of explored blocks
12          self.search_cost = 0 # To keep track of the total search cost
13
14      def is_valid_move(self, x, y):
15          return 1 <= x <= self.rows and 1 <= y <= self.cols and (x, y) not in self.obstacles
16
17      def heuristic(self, x, y):
18          # Define a heuristic function (e.g., Manhattan distance)
19          return abs(x - self.goal[0]) + abs(y - self.goal[1])
20
21      def solve_maze_dfs(self):
22          stack = [(self.start[0], self.start[1])]
23          while stack:
24              x, y = stack.pop()
25              self.search_cost += 1
26              if (x, y) == self.goal:
27                  return True
28              if (x, y) not in self.explored:
29                  self.explored.add((x, y))
30                  # Push valid neighbors onto the stack
31                  for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
32                      nx, ny = x + dx, y + dy

```

```

Lab4_q1.py X
C: > Users > dell > Desktop > NITD > 5th sem > AI > Lab 4_and_5 > Lab4_q1.py > MazeSolver
33             if self.is_valid_move(nx, ny):
34                 stack.append((nx, ny))
35             return False
36
37     def solve_maze_bfs(self):
38         queue = deque([(self.start[0], self.start[1])])
39         while queue:
40             x, y = queue.popleft()
41             self.search_cost += 1
42             if (x, y) == self.goal:
43                 return True
44             if (x, y) not in self.explored:
45                 self.explored.add((x, y))
46                 # Enqueue valid neighbors
47                 for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
48                     nx, ny = x + dx, y + dy
49                     if self.is_valid_move(nx, ny):
50                         queue.append((nx, ny))
51         return False
52
53     def solve_maze_dls(self, limit):
54         stack = [(self.start[0], self.start[1], 0)]
55         while stack:
56             x, y, depth = stack.pop()
57             self.search_cost += 1
58             if depth > limit:
59                 continue # Depth limit reached, backtrack
60             if (x, y) == self.goal:
61                 return True
62             if (x, y) not in self.explored:
63                 self.explored.add((x, y))
64                 # Push valid neighbors onto the stack

```

```
Lab4_q1.py X
C: > Users > dell > Desktop > NITD > 5th sem > AI > Lab 4_and_5 > Lab4_q1.py > MazeSolver
  65     for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
  66         nx, ny = x + dx, y + dy
  67         if self.is_valid_move(nx, ny):
  68             stack.append((nx, ny, depth + 1))
  69     return False
 70 def solve_maze_ucs(self):
 71     # Uniform Cost Search implementation
 72     pq = [(0, self.start[0], self.start[1])]
 73     cost_so_far = {(self.start[0], self.start[1]): 0}
 74
 75     while pq:
 76         current_cost, x, y = heapq.heappop(pq)
 77         self.search_cost += 1
 78
 79         if (x, y) == self.goal:
 80             return True
 81
 82         if (x, y) not in self.explored:
 83             self.explored.add((x, y))
 84
 85         for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
 86             nx, ny = x + dx, y + dy
 87
 88             if self.is_valid_move(nx, ny):
 89                 new_cost = current_cost + 1 # Uniform cost
 90
 91                 if (nx, ny) not in cost_so_far or new_cost < cost_so_far[(nx, ny)]:
 92                     cost_so_far[(nx, ny)] = new_cost
 93                     heapq.heappush(pq, (new_cost, nx, ny))
 94
 95     return False
```

```

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

97     def solve_maze_gbfs(self):
98         # Greedy Best First Search implementation
99         pq = [(self.heuristic(self.start[0], self.start[1]), self.start[0], self.start[1])]
100
101        while pq:
102            _, x, y = heapq.heappop(pq)
103            self.search_cost += 1
104
105            if (x, y) == self.goal:
106                return True
107
108            if (x, y) not in self.explored:
109                self.explored.add((x, y))
110
111                for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
112                    nx, ny = x + dx, y + dy
113
114                    if self.is_valid_move(nx, ny):
115                        heapq.heappush(pq, (self.heuristic(nx, ny), nx, ny))
116
117        return False
118
119    def solve_maze_astar(self):
120        # A* Algorithm implementation
121        pq = [(0, self.start[0], self.start[1])]
122        cost_so_far = {(self.start[0], self.start[1]): 0}
123
124        while pq:
125            current_cost, x, y = heapq.heappop(pq)
126            self.search_cost += 1

```

```
input.txt Lab4_q1.py output.txt
Lab4_q1.py > ...

127
128     if (x, y) == self.goal:
129         return True
130
131     if (x, y) not in self.explored:
132         self.explored.add((x, y))
133
134     for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
135         nx, ny = x + dx, y + dy
136
137         if self.is_valid_move(nx, ny):
138             new_cost = cost_so_far[(x, y)] + 1 # Cost to reach the neighbor
139             if (nx, ny) not in cost_so_far or new_cost < cost_so_far[(nx, ny)]:
140                 cost_so_far[(nx, ny)] = new_cost
141                 priority = new_cost + self.heuristic(nx, ny) # A* priority
142                 heapq.heappush(pq, (priority, nx, ny))
143
144     return False
145
146 def find_solution(self, search_strategy, depth_limit=None):
147     if search_strategy == 'dfs':
148         return self.solve_maze_dfs()
149     elif search_strategy == 'bfs':
150         return self.solve_maze_bfs()
151     elif search_strategy == 'dls':
152         if depth_limit is not None:
153             return self.solve_maze_dls(depth_limit)
154     elif search_strategy == 'ucs':
155         return self.solve_maze_ucs()
156     elif search_strategy == 'gbfs':
157         return self.solve_maze_gbfs()
158     elif search_strategy == 'astar':
```

```

* Lab4_q1.py > ...
159         return self.solve_maze_astar()
160     else:
161         return None
162
163 if __name__ == '__main__':
164     # Increase the recursion limit to a higher value (optional)
165     # sys.setrecursionlimit(10000)
166
167     # Read input from the file
168     with open('input.txt', 'r') as file:
169         rows, cols = map(int, file.readline().split())
170         start_x, start_y, goal_x, goal_y = map(int, file.readline().strip().split())
171         obstacle_lines = file.readline().strip().split()
172         obstacles = {(int(obstacle_lines[i]), int(obstacle_lines[i+1])) for i in range(0, len(obstacle_lines), 2)}
173         search_strategy = file.readline().strip()
174         depth_limit = 3 # You mentioned depth_limit as 3 in your example
175
176     solver = MazeSolver(rows, cols, (start_x, start_y), (goal_x, goal_y), obstacles)
177
178     result = solver.find_solution(search_strategy, depth_limit)
179
180     # Write the result to an output file
181     with open('output.txt', 'w') as file:
182         if result:
183             file.write("Sequence of blocks explored:\n")
184             for block in sorted(solver.explored):
185                 file.write(f"{block[0]} {block[1]}\n")
186             file.write(f"Total search cost: {solver.search_cost}\n")
187         else:
188             file.write("No path found.\n")
189

```

INPUT FILE:

The terminal window shows the following files:

- input.txt**: Contains a 6x6 grid of numbers (1 to 6) and the string "ucs".
- Lab4_q1.py**: The Python script for solving the maze.
- output.txt**: An empty file where the results will be written.

```

input.txt  ×  Lab4_q1.py  output.txt

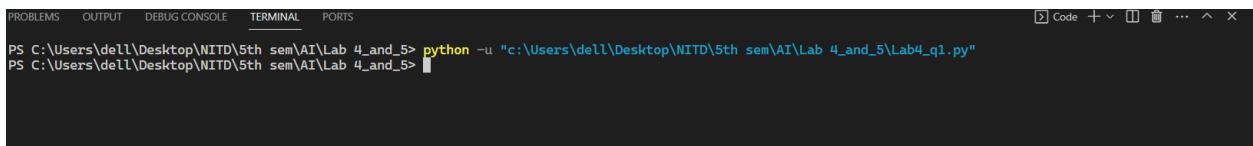
```

```

input.txt
1 6 6
2 1 1 6 6
3 2 1 2 5 3 2 3 3 3 5 4 5 4 6 5 2 5 3 6 3
4 ucs
5

```

OUTPUT file



A screenshot of a terminal window from a code editor. The tabs at the top are PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is selected), and PORTS. On the right side, there are icons for Code, +, ~, ⌂, ⌂, ..., ^, and X. The terminal window shows the following text:

```
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 X
output.txt
1 Sequence of blocks explored:
2 1 1
3 1 2
4 1 3
5 1 4
6 1 5
7 1 6
8 2 2
9 2 3
10 2 4
11 2 6
12 3 1
13 3 4
14 3 6
15 4 1
16 4 2
17 4 3
18 4 4
19 5 1
20 5 4
21 5 5
22 5 6
23 6 4
24 6 5
25 Total search cost: 24
26
```

OBSERVATION:

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

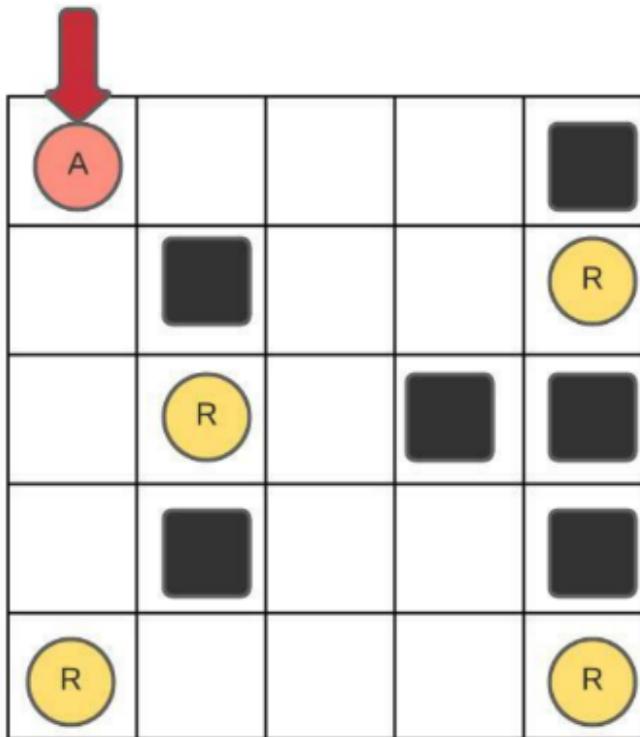
cost of gbfs= 18

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.

Starting Position



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:

- 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))

    # Initialize priority queue with start state
```

```

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)

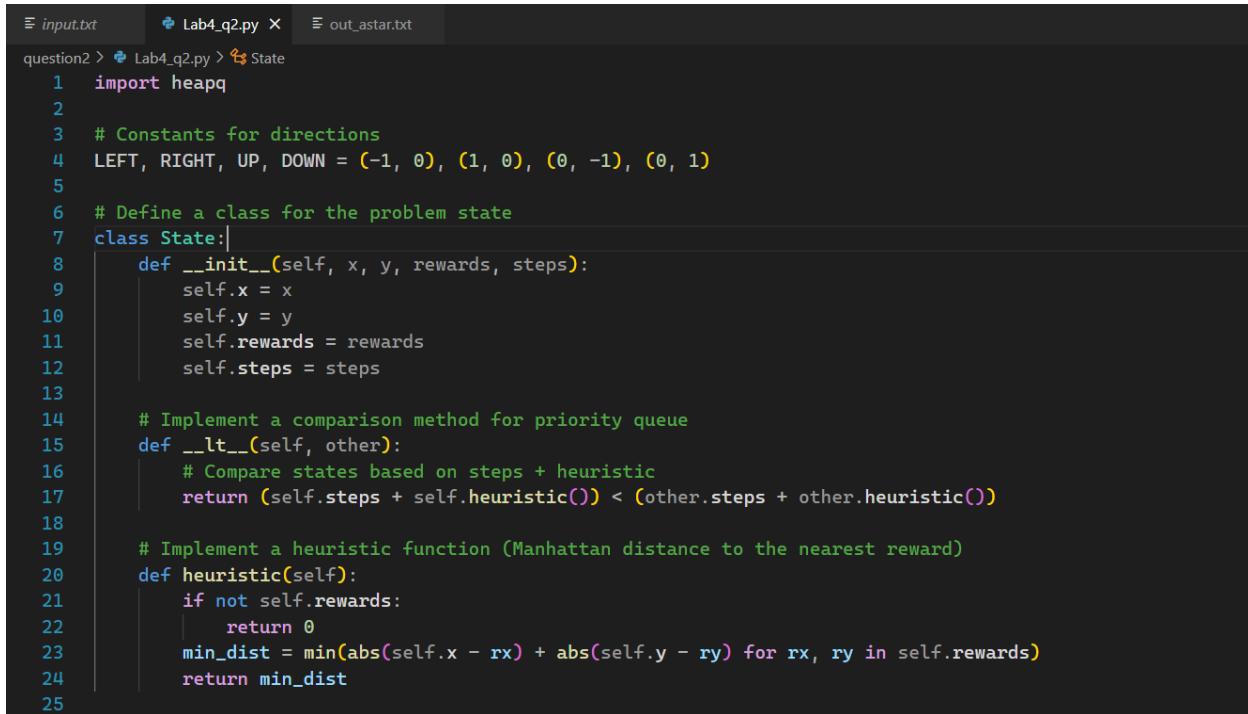
```

```

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

```

CODE SNIPPETS:



The screenshot shows a code editor window with three tabs: 'input.txt', 'Lab4_q2.py', and 'out_astar.txt'. The 'Lab4_q2.py' tab is active and displays the following Python code:

```

question2 > Lab4_q2.py > State
1 import heapq
2
3 # Constants for directions
4 LEFT, RIGHT, UP, DOWN = (-1, 0), (1, 0), (0, -1), (0, 1)
5
6 # Define a class for the problem state
7 class State:
8     def __init__(self, x, y, rewards, steps):
9         self.x = x
10        self.y = y
11        self.rewards = rewards
12        self.steps = steps
13
14    # Implement a comparison method for priority queue
15    def __lt__(self, other):
16        # Compare states based on steps + heuristic
17        return (self.steps + self.heuristic()) < (other.steps + other.heuristic())
18
19    # Implement a heuristic function (Manhattan distance to the nearest reward)
20    def heuristic(self):
21        if not self.rewards:
22            return 0
23        min_dist = min(abs(self.x - rx) + abs(self.y - ry) for rx, ry in self.rewards)
24        return min_dist
25

```

```

26 # Implement the A* algorithm
27 def astar(maze):
28     rows, cols = len(maze), len(maze[0])
29     start_x, start_y = None, None
30     rewards = set()
31
32     # Find the start position and collect rewards
33     for x in range(rows):
34         for y in range(cols):
35             if maze[x][y] == 2:
36                 start_x, start_y = x, y
37             elif maze[x][y] == 3:
38                 rewards.add((x, y))
39
40     start_state = State(start_x, start_y, rewards, 0)
41     priority_queue = [start_state]
42     visited_sequence = [] # Store visited tiles
43
44     while priority_queue:
45         current_state = heapq.heappop(priority_queue)
46         x, y = current_state.x, current_state.y
47
48         if not current_state.rewards:
49             # All rewards collected, exit
50             return current_state.steps, visited_sequence
51
52         visited_sequence.append((x, y)) # Record visited tile
53
54         # Explore possible moves (left, right, up, down)
55         for dx, dy in [(LEFT), (RIGHT), (UP), (DOWN)]:
56             new_x, new_y = x + dx, y + dy

```

```

57
58     if 0 <= new_x < rows and 0 <= new_y < cols and maze[new_x][new_y] != 1:
59         new_rewards = current_state.rewards.copy()
60         if (new_x, new_y) in new_rewards:
61             new_rewards.remove((new_x, new_y))
62
63         new_state = State(new_x, new_y, new_rewards, current_state.steps + 1)
64         heapq.heappush(priority_queue, new_state)
65
66     # No path to collect all rewards
67     return -1, visited_sequence
68
69 # Read input from a text file
70 def read_maze_from_file(filename):
71     with open(filename, 'r') as file:
72         maze = [[int(tile) for tile in line.strip().split()] for line in file.readlines()]
73     return maze
74
75 # Write visited sequence to an output file
76 def write_visited_sequence_to_file(sequence, filename):
77     with open(filename, 'w') as file:
78         for x, y in sequence:
79             file.write(f"{x} {y}\n")
80
81 # Define the input and output filenames
82 input_filename = 'input.txt'
83 output_filename = 'out_astar.txt'
84
85 # Read the maze from the input file
86 maze = read_maze_from_file(input_filename)
87

88 # Run the A* algorithm
89 result, visited_sequence = astar(maze)
90
91 # Write the visited sequence to the output file
92 write_visited_sequence_to_file(visited_sequence, output_filename)
93
94 if result == -1:
95     print("No path to collect all rewards.")
96 else:
97     print(f"Minimum steps to collect all rewards: {result}")
98

```

INPUT File:

The screenshot shows a terminal window with three tabs at the top: 'input.txt' (highlighted), 'Lab4_q2.py', and 'out_astar.txt'. The 'input.txt' tab contains the following text:

```
question2 >  input.txt
1 5 5
2 2 0 0 0 1
3 0 1 0 0 3
4 0 3 0 1 1
5 0 1 0 0 1
6 3 0 0 0 3
7 |
```

2 for start

0 for clear path

3 for reward

1 for obstacle

OUTPUT File:

```
≡ input.txt    ♫ Lab4_q2.py    ≡ out_astar.txt X
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
```

```
question2 > 27 4 0
28 4 0
29 4 0
30 4 0
31 4 0
32 3 0
33 4 0
34
```

```
PS C:\Users\dell\Desktop\NITD\5th sem\AI\Lab 4_and_5\question2> python -u "c:\Users\dell\Desktop\NITD\5th sem\AI\Lab 4_and_5\question2\Lab4_q2.py"
```

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
```

```

❸ search.py > ...
104     while(True):
105
106         # Terminate condition: can't find solution #
107         if stackXY.isEmpty():
108             return []
109
110         # Get informations of current state #
111         xy, path = stackXY.pop() # Take position and path
112         visited.append(xy)
113
114         # Comment this and uncomment 125. This only works for autograder    #
115         # In lectures we check if a state is a goal when we find successors #
116
117         # Terminate condition: reach goal #
118         if problem.isGoalState(xy):
119             return path
120
121         # Get successors of current state #
122         succ = problem.getSuccessors(xy)
123
124         # Add new states in stack and fix their path #
125         if succ:
126             for item in succ:
127                 if item[0] not in visited:
128
129                     # Lectures code:
130                     # All implementations run in autograder and in comments i write
131                     # the proper code that i have been taught in lectures
132                     # if item[0] not in visited and item[0] not in (state[0] for state in stackXY.list):
133                     #     if problem.isGoalState(item[0]):
134                     #         return path + [item[1]]

```

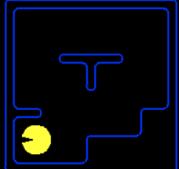


```

134             #     return path + [item[1]]
135
136             newPath = path + [item[1]] # Calculate new path
137             stackXY.push((item[0],newPath))
138
139     util.raiseNotDefined()

```

OUTPUT:



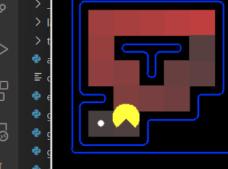
SCORE: 502

```

 85     > ...
 86         ... - DIRECTIONS.LOWER.
 87     return [s, s, w, s, w, w, s, w]
 88
 89 def depthFirstSearch(problem):
 90     """
 91     Search the deepest nodes in the search tree first.
 92
 93     Your search algorithm needs to return a list of actions that reaches the
 94     goal. Make sure to implement a graph search algorithm.
 95
 96     To get started, you might want to try some of these simple commands to
 97     understand the search problem that is being passed in:
 98
 99     print("Start:", problem.getStartState())
100     print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
101     print("Start's successors:", problem.getSuccessors(problem.getStartState()))
102     """
103     from util import Stack
104
105     # stackXY: ((x,y),[path]) #
106
107     PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
108
109     Path found with total cost of 8 in 0.0 seconds
110     Search nodes expanded: 0
111     Pacman emerges victorious! Score: 502
112     Average Score: 502.0
113     Scores: 502.0
114     Win Rate: 1/1 (1.00)
115     Record: Win
116     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
117     [SearchAgent] using function tinyMazeSearch
118     [SearchAgent] using problem type PositionSearchProblem
119     Path found with total cost of 8 in 0.0 seconds
120     Search nodes expanded: 0
121     Pacman emerges victorious! Score: 502
122
  
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

OUTLINE TIMELINE



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 implementations 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))

    PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
    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
    
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

powershell python

OUTLINE TIMELINE



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:

```
❸ search.py  x
❹ search.py > ...
141 def breadthFirstSearch(problem):
142     """Search the shallowest nodes in the search tree first."""
143     from util import Queue
144
145     # queueXY: ((x,y),[path]) #
146     queueXY = Queue()
147
148     visited = [] # Visited states
149     path = [] # Every state keeps it's path from the starting state
150
151     # Check if initial state is goal state #
152     if problem.isGoalState(problem.getStartState()):
153         return []
154
155     # Start from the begin (function) getStartState: Any is empty list #
156     queueXY.push((problem.getStartState(),[]))
157
158     while(True):
159
160         # Terminate condition: can't find solution #
161         if queueXY.isEmpty():
162             return []
163
164         # Get informations of current state #
165         xy,path = queueXY.pop() # Take position and path
166         visited.append(xy)
167
168         # Comment this and uncomment 179. This is only works for autograder
```

```

168     # Comment this and uncomment 179. This is only works for autograder
169     # In lectures we check if a state is a goal when we find successors
170
171     # Terminate condition: reach goal #
172     if problem.isGoalState(xy):
173         return path
174
175     # Get successors of current state #
176     succ = problem.getSuccessors(xy)
177
178     # Add new states in queue and fix their path #
179     if succ:
180         for item in succ:
181             if item[0] not in visited and item[0] not in (state[0] for state in queueXY.list):
182
183                 # Lectures code:
184                 # All implementations run in autograder and in comments i write
185                 # the proper code that i have been taught in lectures
186                 # if problem.isGoalState(item[0]):
187                 #     return path + [item[1]]
188
189                 newPath = path + [item[1]] # Calculate new path
190                 queueXY.push((item[0],newPath))
191
192     util.raiseNotDefined()
193

```

OUTPUT:



View Go Run Terminal Help ← → , LA_4_5.search

... search.py x
search.py > ...

```
170  
171     # Terminate condition: reach goal #  
172     if problem.isGoalState(xy):  
173         return path  
174  
175     # Get successors of current state #  
176     succ = problem.getSuccessors(xy)  
177  
178     # Add new states in queue and fix their path #  
179     if succ:  
180         for item in succ:  
181             if item[0] not in visited and item[0] not in Q:  
182                 # Lectures code:  
183                 # All implementations run in autograder and  
184                 # the proper code that i have been taught  
185                 # if problem.isGoalState(item[0]):  
186  
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
```

PS C:\Users\dell\Downloads\LA_4_5_search-20230908T112728Z-001\LA_4_5_search> python -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores: 442.0
Win Rate: 1/1 (1.00)
Record: Win
PS C:\Users\dell\Downloads\LA_4_5_search-20230908T112728Z-001\LA_4_5_search> python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620

CS188 Pacman

SCORE: -44

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:

```

193
194 def uniformCostSearch(problem):
195     """Search the node of least total cost first."""
196     from util import PriorityQueue
197
198     # queueXY: ((x,y),[path],priority) #
199     queueXY = PriorityQueue()
200
201     visited = [] # Visited states
202     path = [] # Every state keeps it's path from the starting state
203
204     # Check if initial state is goal state #
205     if problem.isGoalState(problem.getStartState()):
206         return []
207
208     # Start from the beginning and find a solution, path is empty list #
209     # with the cheapest priority
210     queueXY.push((problem.getStartState(),[],0))
211
212     while(True):
213
214         # Terminate condition: can't find solution #
215         if queueXY.isEmpty():
216             return []
217
218         # Get informations of current state #
219         xy, path = queueXY.pop() # Take position and path
220         visited.append(xy)
221
222         # This only works for autograder #
223         # In lectures we check if a state is a goal when we find successors #

```

search.py

```

224
225     # Terminate condition: reach goal #
226     if problem.isGoalState(xy):
227         return path
228
229     # Get successors of current state #
230     succ = problem.getSuccessors(xy)
231
232     # Add new states in queue and fix their path #
233     if succ:
234         for item in succ:
235             if item[0] not in visited and (item[0] not in (state[2][0] for state in queueXY.heap)):
236
237                 # Like previous algorithms: we should check in this point if successor
238                 # is a goal state so as to follow lectures code
239
240                 newPath = path + [item[1]]
241                 pri = problem.getCostOfActions(newPath)
242
243                 queueXY.push((item[0],newPath),pri)
244
245             # State is in queue. Check if current path is cheaper from the previous one #
246             elif item[0] not in visited and (item[0] in (state[2][0] for state in queueXY.heap)):
247                 for state in queueXY.heap:
248                     if state[2][0] == item[0]:
249                         oldPri = problem.getCostOfActions(state[2][1])
250
251                         newPri = problem.getCostOfActions(path + [item[1]])
252
253                         # State is cheaper with his new father -> update and fix parent #
254                         if oldPri > newPri:
255                             newPath = path + [item[1]]

```

OUTPUT:

The terminal window displays the following output:

```
PS C:\Users\dell\Downloads\LA_4_5_search-20230908T112100Z-001\CS188 Pacman
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
```

The right side of the window shows a Pacman maze with a blue path drawn on it. A yellow circle represents the ghost. The score is displayed as **SCORE: -33**.

The terminal window displays the following output:

```
PS C:\Users\dell\Downloads\LA_4_5_search-20230908T112100Z-001\CS188 Pacman
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
```

The right side of the window shows a Pacman maze with a blue path drawn on it. A yellow circle represents the ghost. The score is displayed as **SCORE: 94**.



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:

```

259 def nullHeuristic(state, problem=None):
260     """
261     A heuristic function estimates the cost from the current state to the nearest
262     goal in the provided SearchProblem. This heuristic is trivial.
263     """
264     return 0
265
266 from util import PriorityQueue
267 class MyPriorityQueueWithFunction(PriorityQueue):
268     """
269     Implements a priority queue with the same push/pop signature of the
270     Queue and the Stack classes. This is designed for drop-in replacement for
271     those two classes. The caller has to provide a priority function, which
272     extracts each item's priority.
273     """
274     def __init__(self, problem, priorityFunction):
275         "priorityFunction (item) -> priority"
276         self.priorityFunction = priorityFunction      # store the priority function
277         PriorityQueue.__init__(self)          # super-class initializer
278         self.problem = problem
279     def push(self, item, heuristic):
280         "Adds an item to the queue with priority from the priority function"
281         PriorityQueue.push(self, item, self.priorityFunction(self.problem, item, heuristic))
282
283     # Calculate f(n) = g(n) + h(n) #
284     def f(problem, state, heuristic):
285

```

```

286         return problem.getCostOfActions(state[1]) + heuristic(state[0],problem)
287
288     def aStarSearch(problem, heuristic=nullHeuristic):
289         """Search the node that has the lowest combined cost and heuristic first."""
290         # queueXY: ((x,y),[path]) #
291         queueXY = MyPriorityQueueWithFunction(problem,f)
292
293         path = [] # Every state keeps it's path from the starting state
294         visited = [] # Visited states
295
296         # Check if initial stat (parameter) problem: Any
297         if problem.isGoalState(problem.getStartState()):
298             return []
299
300         # Add initial state. Path is an empty list #
301         element = (problem.getStartState(),[])
302
303         queueXY.push(element,heuristic)
304
305         while(True):
306
307             # Terminate condition: can't find solution #
308             if queueXY.isEmpty():
309                 return []
310

```

```

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

```

OUTPUT

The terminal window displays the `search.py` script and its execution results. The script includes implementations for Breadth-First Search (BFS), Depth-First Search (DFS), A* search, and Uniform Cost Search (UCS). The execution output shows a win rate of 1/1 (1.00), a record of Win, and details about the search process like heuristic type and path length.

```

search.py x
search.py > ...
330     if succ:
331         for item in succ:
332             if item[0] not in visited:
333
334                 # Like previous algorithms: we s
335                 # is a goal state so as to follo
336
337                 newPath = path + [item[1]] # Fix
338                 element = (item[0],newPath)
339                 queueXY.push(element,heuristic)
340     util.raiseNotDefined()
341
342
343 # Abbreviations
344 bfs = breadthFirstSearch
345 dfs = depthFirstSearch
346 astar = aStarSearch
347 ucs = uniformCostSearch
348 # Done by rex (211210053)
349 # Spent 12 hours was very tough no kidding

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Win Rate: 1/1 (1.00)
Record: Win
PS C:\Users\dell\Downloads\LA_4_5_search-20230908T112728Z-001\LA_4_
uristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 549

```

SCORE: -67

The right side of the terminal window shows a complex maze with a blue path from the start to the goal. The path is highlighted in blue, and the score is displayed as **SCORE: -67**.

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:

```

372 def cornersHeuristic(state, problem):
373     """
374     A heuristic for the CornersProblem that you defined.
375
376     state: The current search state
377         (a data structure you chose in your search problem)
378
379     problem: The CornersProblem instance for this layout.
380
381     This function should always return a number that is a lower bound on the
382     shortest path from the state to a goal of the problem; i.e. it should be
383     admissible (as well as consistent).
384     """
385     corners = problem.corners # These are the corner coordinates
386     walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
387
388     from util import manhattanDistance
389
390     # Goal state #
391     if problem.isGoalState(state):
392         return 0
393
394     else:
395         distancesFromGoals = [] # Calculate all distances from goals(not visited corners)
396
397         for index,item in enumerate(state[1]):
398             if item == 0: # Not visited corner
399                 # Use manhattan method #
400                 distancesFromGoals.append(manhattanDistance(state[0],corners[index]))
401
402                 distancesFromGoals.append(manhattanDistance(state[0],corners[index]))
403
404         # Worst case. This guess should be higher than real. Pick higher distance #
405         return max(distancesFromGoals)
406
407     return 0 # Default to trivial solution

```

OUTPUT:



SCORE: 5

```
py > 🐕 CornersProblem

    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

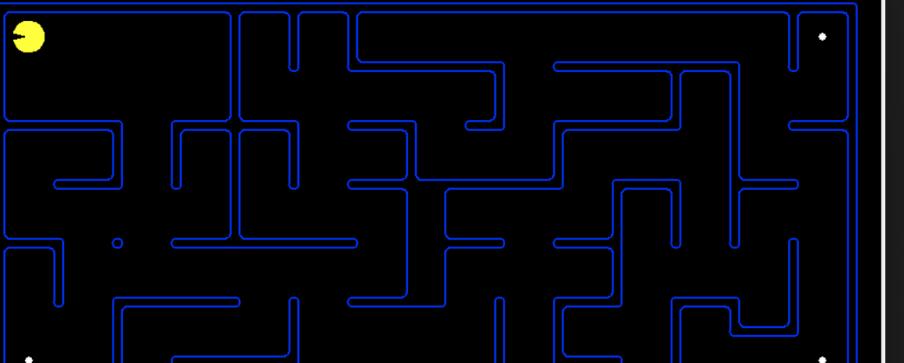
s AStarCornersAgent(SearchAgent):
    "A SearchAgent for FoodSearchProblem using A* and your foodHeuristic"
def __init__(self):
    self.searchFunction = lambda prob: search.aStarSearch(prob, cornersHeuristic)
    self.searchType = CornersProblem

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

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
```

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

[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds



Path found with total cost: 106
Search nodes expanded: 1966
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores: 512.0
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 mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 1966
```

▼ Question 4.6 (3 points): Corners Problem: Heuristic

Note: Make sure to complete Question 4 before working on Question 6, because Question 6 builds

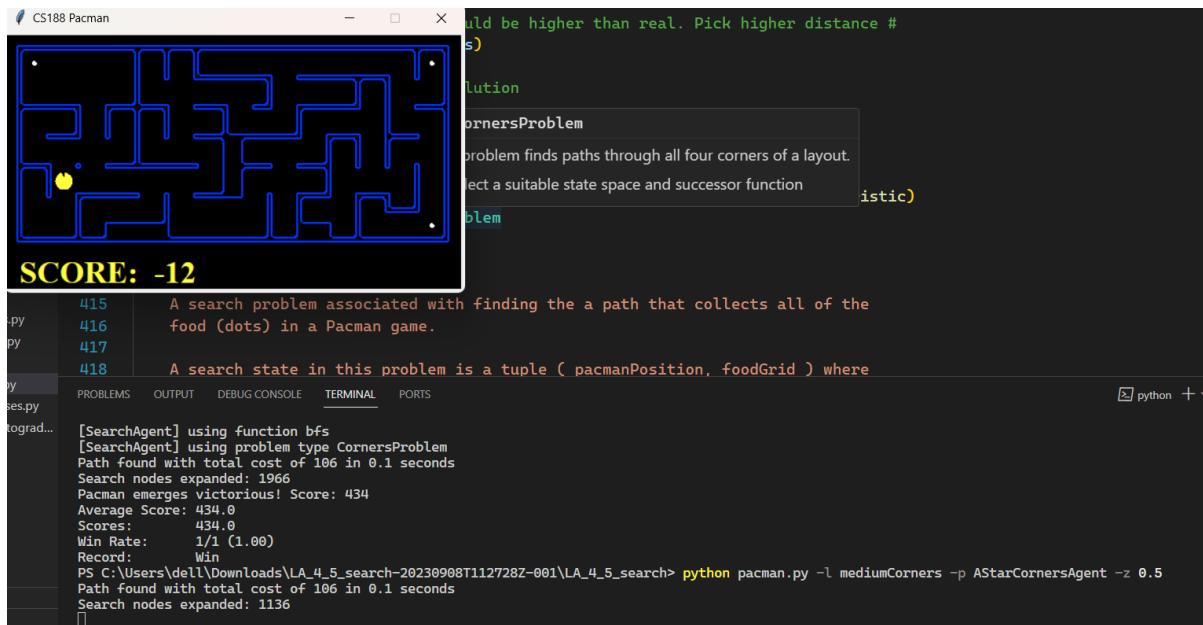
Upon your answer for Question 4.

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:

```

searchAgents.py X
CornersProblem

413 class FoodSearchProblem:
414     """
415         A search problem associated with finding the a path that collects all of the
416         food (dots) in a Pacman game.
417
418         A search state in this problem is a tuple ( pacmanPosition, foodGrid ) where
419             pacmanPosition: a tuple (x,y) of integers specifying Pacman's position
420             foodGrid:      a Grid (see game.py) of either True or False, specifying remaining food
421     """
422     def __init__(self, startingGameState):
423         self.start = (startingGameState.getPacmanPosition(), startingGameState.getFood())
424         self.walls = startingGameState.getWalls()
425         self.startingGameState = startingGameState
426         self._expanded = 0 # DO NOT CHANGE
427         self.heuristicInfo = {} # A dictionary for the heuristic to store information
428
429     def getStartState(self):
430         return self.start
431
432     def isGoalState(self, state):
433         return state[1].count() == 0
434
435     def getSuccessors(self, state):
436         """Returns successor states, the actions they require, and a cost of 1."""
437         successors = []
438         self._expanded += 1 # DO NOT CHANGE
439         for direction in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
440             x,y = state[0]
441             dx, dy = Actions.directionToVector(direction)
442             nextx, nexty = int(x + dx), int(y + dy)
443             if not self.walls[nextx][nexty]:
444                 nextFood = state[1].copy()
445                 nextFood[nextx][nexty] = False
446                 successors.append( ((nextx, nexty), nextFood, direction, 1) )
447
448         return successors
449
450     def getCostOfActions(self, actions):
451         """Returns the cost of a particular sequence of actions. If those actions
452         include an illegal move, return 999999"""
453         x,y= self.getStartState()[0]
454         cost = 0
455         for action in actions:
456             # figure out the next state and see whether it's legal
457             dx, dy = Actions.directionToVector(action)
458             x, y = int(x + dx), int(y + dy)
459             if self.walls[x][y]:
460                 return 999999
461             cost += 1
462         return cost

```

OUTPUT:

```

searchAgents.py > CornersProblem
439     for direction in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
440         x,y = state[0]
441         dx, dy = Actions.directionToVector(direction)
442         nextx, nexty = int(x + dx), int(y + dy)
443         if not self.walls[nextx][nexty]:
444             nextFood = state[1].copy()
445             nextFood[nextx][nexty] = False
446             successors.append( ( (nextx, nexty), nextFood, direction, 1) )
447     return successors
448
449     def getCostOfActions(self, actions):
450         """Returns the cost of a particular sequence of actions. If those actions
451         Record: Win
452         PS C:\Users\dell\Downloads\LA_4_5_search-20230908T112728Z-001\LA_4_5_search> python pacman.py -l testSearch -p AStarFoodSearchAgent
453         Path found with total cost of 7 in 0.0 seconds
454         Search nodes expanded: 10
455
456

```

```

searchAgents.py > CornersProblem
439     for direction in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
440         x,y = state[0]
441         dx, dy = Actions.directionToVector(direction)
442         nextx, nexty = int(x + dx), int(y + dy)
443         if not self.walls[nextx][nexty]:
444             nextFood = state[1].copy()
445             nextFood[nextx][nexty] = False
446             successors.append( ( (nextx, nexty), nextFood, direction, 1) )
447     return successors
448
449     def getCostOfActions(self, actions):
450         """Returns the cost of a particular sequence of actions. If those actions
451         Record: Win
452         PS C:\Users\dell\Downloads\LA_4_5_search-20230908T112728Z-001\LA_4_5_search> python pacman.py -l trickySearch -p AStarFoodSearchAgent
453         Path found with total cost of 60 in 7.6 seconds
454         Search nodes expanded: 4299
455
456

```

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.

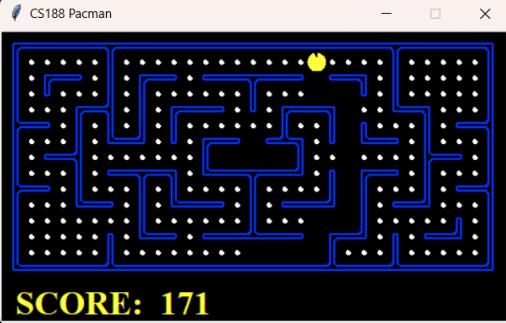
CODE:

```

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



```

40     gameState.
41     """
42     # Here are some useful elements of the startState
43     startPosition = gameState.getPacmanPosition()
44     food = gameState.getFood()
45     walls = gameState.getWalls()
46     problem = AnyFoodSearchProblem()
47
48     temp = food.asList()
49
50     from search import *
51
52     # Bfs finds closest food first.
53     return breadthFirstSearch(problem)
54     util.raiseNotDefined()
55
56 class AnyFoodSearchProblem(PositionSearchProblem):
57     """
58         A search problem for finding a path to any food.

```

SCORE: 171

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS python +

S C:\Users\dell\Downloads\LA_4_5_search-20230908T112728Z-001\LA_4_5_search> python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

SearchAgent] using function depthFirstSearch

SearchAgent] using problem type PositionSearchProblem

path found with cost 350.

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> █
```