MAS – Ex1

Erfan Moosavi Monazzah - 401722199

April 5, 2023

Introduction:

The vacuum agent problem involves designing an agent that can navigate a rectangular room and clean all the dirt present in it. The agent is equipped with sensors that allow it to detect the presence of dirt in the tile it howered. The agent can perform four actions - move forward, rotate 90 degrees, check for dirt, and suck dirt.

Problem Statement:

The task of the vacuum agent is to clean the entire room while minimizing the number of movements and actions taken. The agent should be able to move efficiently and avoid unnecessary movements or revisits to already cleaned areas.

Proposed Solution:

Our solution to the vacuum agent problem is based on a simple yet effective strategy. The agent will start by rotating until it faces the nearest wall. Once facing the wall, the agent will move toward the wall for one tile. It again checks for the nearest wall and rotate toward it. It sucks up all the dirt along the way. This process will continue until the entire room is cleaned.

Algorithm:

1. Start facing a random direction in a random tile
2. Check for dirt
3. If dirt is present, suck it
4. Rotate until the agent is facing the nearest wall
5. Move toward the wall for one tile
6. Repeat steps 2-6 until the entire room is cleaned (all tile traversed)

Evaluation:

We evaluated our proposed solution by running simulations of the agent in various room configurations. The results show that our solution is effective and efficient in cleaning the entire room. The agent avoids revisiting cleaned areas, and the number of movements and actions taken are minimized. Since we can not detect neighbor tile's dirt, we need to traverse all tiles.

You can visit the attached GIF files to this report to better understand the algorithm:
Brown squares: Dirt
White squares: Clean
Bot: Vacuum Cleaner Agent
In this picture, the agent is facing north.

Analysis of Vacuum Cleaner Simulation Results

We ran the simulation of the vacuum cleaner problem three times, each time with different inputs. The cost of sucking a dirty tile is 2 and the cost of moving to a new tile is 1. No cost is defined for rotating.

The results of the simulations are as follows:

Simulation 1:
Dirty Tiles Generated: 9
Sucked Tiles: 9
Total Cost: 34

Simulation 2:
Dirty Tiles Generated: 7
Sucked Tiles: 7
Total Cost: 30

Simulation 3:
Dirty Tiles Generated: 6
Sucked Tiles: 6
Total Cost: 28

We can see that the number of dirty tiles generated and the number of tiles cleaned are roughly the same across all three simulations. However, the total cost of the cleaning process varies significantly. Simulation 3 has the lowest total cost, while Simulation 1 has the highest.

On average, the simulation results are as follows:
Average Dirty Tiles Generated: 7.33
Average Sucked Tiles: 7.33
Average Total Cost: 30.67

Based on the results of the simulations, we can conclude that the algorithm used to solve the vacuum cleaner problem is effective in terms of cleaning all the dirty tiles generated. However, the total cost of the cleaning process varies depending on the initial configuration of the tiles.

Codes: Agent

```
class VacuumCleanerAgent:
    def __init__(self, board_shape=(4,4), dirty_tile_ratio=0.4, starting_tile=None,
starting_direction=None):
```

The VacuumCleanerAgent class is an implementation of a simple vacuum cleaner agent that moves around a grid and cleans dirty tiles. The __init__ method initializes the agent with a board of a given shape and ratio of dirty tiles, and sets the starting tile and direction.

```
starting_tile = starting_tile if starting_tile is not None else (random.randint(0, board_shape[0]-1),
random.randint(0, board_shape[1]-1))
starting_direction = starting_direction if starting_direction is not None else random.choice('news')
```

If starting_tile or starting_direction is not provided, random values are generated.

```
self.arrows = {'n':'▲', 'e':'  ', 'w':'  ', 's':'▼'}
self.dirty_arrows = {'n':'↑↑', 'e':'⇉', 'w':'⇇', 's':'↓↓'}
self.next_direction = {'n':'e', 'e':'s', 's':'w', 'w':'n'}
self.clear_tile = '□'
self.traversed_tile = '  '
self.dirty_tile = 'x'
```

These dictionaries define various symbols used to represent the tiles and the direction of the agent.

```
self.init_board(board_shape, dirty_tile_ratio, starting_tile, starting_direction)
self.cost = 0
```

The init_board method initializes the board with dirty tiles and sets the cost to 0.

```
def init_board(self, board_shape, dirty_tile_ratio, starting_tile, starting_direction):
    self.tile = starting_tile
    self.direction = starting_direction
    self.arrow = self.arrows[self.direction]

    self.board_shape = board_shape
    self.board = [[None for _ in range(board_shape[1])] for _ in range(board_shape[0])]
    self.board[self.tile[0]][self.tile[1]] = self.arrow
    self.dirty_tiles_num = 0
    self.sucked_tiles_num = 0

    for i in range(board_shape[0]):
        for j in range(board_shape[1]):
            if self.board[i][j] == self.arrow:
                continue
            self.board[i][j] = self.dirty_tile if random.random()<= dirty_tile_ratio else self.clear_tile
            if self.board[i][j] == self.dirty_tile:
                self.dirty_tiles_num += 1
```

The init_board method initializes the board, sets the starting tile and direction, and calculates the number of dirty tiles.

```
def draw_board(self):
    pprint(self.board)
    print(self.tile)
    print()
    time.sleep(0.5)
```
The draw_board method prints the board and the current tile of the agent, and pauses for half a second.

```
def board_size(self):
    return self.board_shape[0] * self.board_shape[1]
```
The board_size method returns the total number of tiles on the board.

next_tile(self, curr_tile, direction): This function takes two arguments, curr_tile (a tuple representing the current tile the robot is on) and direction (a string representing the direction the robot should move in). It uses a switch case statement to determine the next tile the robot should move to based on the current tile and the direction it needs to move in. The next tile is returned as a tuple.

move(self): This function updates the board with the robot's current position by replacing the tile at the current position with either the dirty or traversed tile, depending on whether the tile is dirty or not. The robot's position is then updated to the next tile using the next_tile function. The function then updates the robot's arrow based on the new tile it is on, and finally updates the cost of moving by 1.

turn_right(self): This function updates the robot's direction by changing it to the next direction in a clockwise direction. It then updates the arrow to be either a clean or dirty arrow based on the new direction, depending on whether the current tile is dirty or not. Finally, the function updates the board with the new arrow.

dirt_detected(self): This function returns a boolean value indicating whether the current tile is dirty or not. It does this by checking if the arrow currently being used is a dirty arrow.

suck(self): This function updates the board with a clean arrow at the current position and increments the cost by 2. It also increments the number of tiles that have been cleaned by 1.

Codes: Algorithm

The solve_tttdwlt function is designed to solve the "Turning To The Direction With Less Tiles" problem for a vacuum cleaner agent. Here is a breakdown of each line of the function:

def solve_tttdwlt(agent: VacuumCleanerAgent):
The function takes a VacuumCleanerAgent object as input, which represents the vacuum cleaner agent on a board.

traversed_tiles = []
Create an empty list called traversed_tiles, which will be used to store the tiles that the vacuum cleaner agent has already traversed.

while len(traversed_tiles) < agent.board_size():
Start a while loop that continues until the number of traversed tiles is equal to the total number of tiles on the board.

curr_tile = agent.tile
traversed_tiles.append(curr_tile)
agent.draw_board()
Get the current tile that the vacuum cleaner agent is on and append it to the traversed_tiles list. Then, draw the board with the current state of the agent.

```
if agent.dirt_detected():
    agent.suck()
    agent.draw_board()
```

Check if the current tile is dirty. If it is, call the suck() method of the agent to clean the tile and update the board. Then, draw the board with the updated state.

```
candidate_directions = {
    'n': curr_tile[0],
    'e': agent.board_shape[1] - 1 - curr_tile[1],
    'w': curr_tile[1],
    's': agent.board_shape[0] - 1 - curr_tile[0]
    }
```

candidate_directions = {k:v for k,v in candidate_directions.items() if v != 0}
Create a dictionary called candidate_directions that maps the four possible directions (north, east, west, south) to the number of tiles in that direction. The number of tiles is calculated by finding the difference between the current tile's coordinates and the edge of the board in that direction. Then, remove any directions with 0 tiles from the dictionary.

```
selected_direction = None
while len(candidate_directions)>0:
    selected_direction = min(candidate_directions, key=candidate_directions.get)
    next_tile = agent.next_tile(curr_tile, selected_direction)
    if next_tile not in traversed_tiles:
        break
    del candidate_directions[selected_direction]
```

Initialize a variable called selected_direction to None. Then, start a while loop that continues until there are no more candidate directions to choose from. The loop selects the direction with the fewest tiles from the candidate_directions dictionary using the min() function. It then calculates the next tile in that direction using the next_tile() method of the agent and checks if it has already been traversed. If it has not been traversed, the loop breaks. Otherwise, the direction is removed from the candidate_directions dictionary using the del keyword.

```
if selected_direction == None:
    break
```

If no direction was selected in the previous step (i.e., all candidate directions have already been traversed), break the loop.

```
while agent.direction != selected_direction:
    agent.turn_right()
    agent.draw_board()
```

Turn the vacuum cleaner agent in the direction selected in the previous step by calling the turn_right() method repeatedly until it is facing the desired direction. Draw the board with the updated state after each turn.

```
agent.move()
```

Move the vacuum cleaner agent one tile in the direction it is facing.