Sanjee Yogeswaran

Midterm Coding Assignment – RoboVac

Discussion for no obstacles provided (RoboVac1):
The strategy for moving the RoboVac without a pre-provided list of obstacles is based off the A* search. With obstacles, it is an A* search with Manhattan Distance as a heuristic. However, when no obstacles are provided, the algorithm can't distinguish between an unvisited block, and furniture. We can avoid this problem by storing the previous state(coordinates) of the RoboVac. If it does not change, furniture is in the way, and so add the coordinate of the furniture to a list holding obstacles.

The normal A* aspect, lower in the code, is used to determine which adjacent block is closest in Manhattan distance to the RoboVac is both not in the obstacle list, and also in the unvisited list. If these conditions are met, the RoboVac moves to the position that is in the unvisited list. If these conditions aren't met, the RoboVac moves in a random direction that is not an obstacle.

However, if the RoboVac visits blocks that are not unvisited, that is, if the RoboVac continually visits a previously visited spot, the consecutive visited counter is incremented each time. After it reaches a threshold of 8 moves, seek mode is enabled.

Seek Mode:
Seek mode involves selecting a random corner that is unvisited or a random block in the unvisited list. The algorithm then gives RoboVac 8 moves to get to an unvisited block closer to the seek target, otherwise it selects a new seek target(unvisited corner or random in unvisited list). Here two heuristics are used, heuristic() which calculates the distance of the 4 adjacent locations of the RoboVac against all unvisited blocks. In seek mode, only those with distance/priority 0 are selected, meaning that there literally has to be an unvisited block directly adjacent to the RoboVac to be selected. Otherwise heuristic2 is used with the 4 adjacent locations of the RoboVac against a single target block, the random unvisited corner or random unvisited block in the unvisited list. By randomly going through seek coordinates every 8 moves, the RoboVac is highly likely to encounter an unvisited block on the way. As soon as the RoboVac hits an unvisited block, seek mode is disabled, the consecutive visited counter is reset and regular path-finding resumes.

If the regular path-finding does not work because the RoboVac is in a sea of green visited blocks, it will take 8 consecutive moves to these already visited blocks to re-enable seek mode, with a new target this time.

Loop Avoider:
Sometimes though, an obstacle will be in the way, and the RoboVac loops between two positions against an obstacle or wall. This is where the Loop Avoider takes effect. The values of how many times RoboVac has landed on a particular spot is stored in the self.loop_avoider dictionary (key=coordinate tuple, value=count). The Loop Avoider is activated when the RoboVac lands on a spot that has been visited 4 times or more already.
In this mode, a random direction that does not result in a wall or obstacle for the RoboVac is returned and added to the self.loop_avoider_queue 3 times. This queue is drained early in the get_next_move code, and allows the RoboVac to move in a certain direction 4 times total, hopefully clearing the obstacle(which are usually max 4 blocks long). This 'reset' of position allows the heuristics to reroute better.

Conclusion:
This mix of using the A* search algorithm as first choice, seeking a random corner/random unvisited block as second choice, and then finally moving in a random direction 4 times if the second choice is taking too many cycles, provides usually good enough performance to get 100% coverage. There are however some certain situations, such as a furniture block that is not accessible by the RoboVac to be registered as an unvisited block. The RoboVac then spends many cycles trying to move to this inaccessible location, and neglects other unvisited blocks. Due to seek_mode selecting a new target at random every 8 moves, this is an unlikely situation, but possible if there is a

single or a couple of blocks unvisited, as statistically fewer unvisited blocks means a lower chance of a true unvisited block being the target of seek mode over a block of inaccessible furniture. While the algorithm is not guaranteed to have 100% coverage, it almost always gets above 0.95 coverage, due to the statistically likely nature of seek mode selecting a true unvisited block when there are many of them.

Discussion for obstacles provided (RoboVac0):
Here the A* algorithm is used, finding the shortest distance to an unvisited block that is not in the obstacle list, using Manhattan distance as the heuristic. The obstacle list is pre-populated with all blocks provided from the config_list. The RoboVac will always move to an adjacent unvisited block, and then start moving randomly if there are no adjacent unvisited blocks. Due to knowing all obstacle positions, the algorithm does not have to deal with furniture that is mistaken for an unvisited block, generally improving the efficiency of this approach compared with detecting obstacles on the fly like the previous code.

Results (using block list)

Level	Coverage	Max Tiles	Cycles Used	Efficiency	Filename
0	100	182	187	0.97	RoboVac0.py
1	100	164	169	0.97	RoboVac0.py
2	100	146	152	0.96	RoboVac0.py
3	100	150	167	0.90	RoboVac0.py
4	100	145	170	0.85	RoboVac0.py
5	100	156	193	0.81	RoboVac0.py

Results (No obstacles pre-provided)

Level	Coverage	Max Tiles	Cycles Used	Efficiency	Filename
0	100	168	173	0.97	RoboVac1.py
1	100	164	172	0.95	RoboVac1.py
2	100	159	186	0.85	RoboVac1.py
3	100	149	198	0.75	RoboVac1.py
4	100	159	209	0.76	RoboVac1.py
5	100	143	221	0.65	RoboVac1.py

No obstacles provided code:

```
"""
create robot vacuum that cleans all the floors of a grid.
main creates an instance of RoboVac (your code) and provides:
- grid size
- loc of robovac
- list of x,y,w,h tuples are instance of rectangluar blocks

goal: visit all tiles
exec will : create instance and in game loop call : nextMove()  ??
"""
import random
from queue import PriorityQueue
class RoboVac:
    def __init__(self, config_list):
        self.room_width, self.room_height = config_list[0]
        self.pos = config_list[1]  # starting position of vacuum
```

```python
        self.current_pos = (self.pos[0], self.pos[1])
        # Block list not used (config.list[2])

        self.obstacles = set()
        self.unvisited_blocks = set()
        self.priority_queue = PriorityQueue()
        # Creates unvisited blocks set; adds every block in grid to set
        self.initialize_unvisited_blocks()

        # Holds previous state
        self.prev_pos = (-1, -1)
        self.prev_direction = -1

        # Loop Avoider: how many times RoboVac has visited a block
        # (key: position, value: count of times visited)
        self.loop_avoider = {}
        # Consecutive Visited: how many times RoboVac has consecutively
        # visited already visited blocks
        self.consecutive_visited = 0
        # Seek Mode: sets mode to seek a corner or random unvisited block
        self.seek_mode = False
        # Next Unvisited: position of corner or random unvisited block
        self.next_unvisited = (-1, -1)
        # Loop avoi used to move RoboVac in certain direction multiple times
        self.loop_avoider_queue = PriorityQueue()
        # Adds positions of all walls to obstacle list
        self.initialize_walls()

        # fill in with your info
        self.name = "Sanjee Yogeswaran"
        self.id = "47514289"

    ########################################################################
    # Creates a set with all blocks in the grid given width and height
    ########################################################################
    def initialize_unvisited_blocks(self):
        # Initialize the set of unvisited blocks
        for x in range(self.room_width):
            for y in range(self.room_height):
                self.unvisited_blocks.add((x, y))

    def initialize_walls(self):
        for x in range(self.room_width):
            self.obstacles.add((x, self.room_height))
            self.obstacles.add((x, -1))
        for y in range(self.room_height):
            self.obstacles.add((-1, y))
            self.obstacles.add((self.room_width, y))
```

```python
########################################################################
# A* heuristic using Manhattan distance
# This algorithm checks against all unvisited blocks
########################################################################
def heuristic(self, position):
    # Heuristic: Manhattan distance to the closest unvisited block
    min_distance = float("inf")
    for block in self.unvisited_blocks:
        distance = abs(position[0] - block[0]) + abs(position[1] - block[1])
        min_distance = min(min_distance, distance)
    return min_distance

########################################################################
# Modified heuristic using Manhattan distance
# This algorithm checks against a specific block
# Param target_block is expected to be a random unvisited block or corner
########################################################################
def heuristic2(self, position, target_block):
    # Heuristic: Manhattan distance to a target block
    distance = abs(position[0] - target_block[0]) + abs(
        position[1] - target_block[1]
    )
    return distance

def get_next_move(self, current_pos):
    # Define possible directions: north, east, south, west
    directions = [(0, -1), (1, 0), (0, 1), (-1, 0)]
    # Define coordinates for all four corners
    corners = [
        (0, 0),
        (0, self.room_width - 1),
        (self.room_height - 1, 0),
        (self.room_width - 1, self.room_height - 1),
    ]

    # Queue used in loop avoider algorithm
    # RoboVac moves 4 times in a certain direction to avoid obstacles
    if not self.loop_avoider_queue.empty():
        priority, next_pos, direction = self.loop_avoider_queue.get()
        if next_pos in self.unvisited_blocks:
            self.unvisited_blocks.remove(next_pos)
            self.consecutive_visited = 0
            self.loop_avoider_queue = PriorityQueue()
            self.seek_mode = False
        # Keep counter of consecutive moves to visited blocks
        else:
            self.consecutive_visited += 1
        self.prev_pos = next_pos
        self.prev_direction = direction
        return direction
    # Logic if RoboVac has not moved(hit an obstacle or wall)
    if self.prev_pos == current_pos:
        # Detect obstacles on the fly if RoboVac didn't move position
        obstacle = (
            self.prev_pos[0] + directions[self.prev_direction][0],
            self.prev_pos[1] + directions[self.prev_direction][1],
        )
        print(f"added obstacle {obstacle}")
        self.obstacles.add(obstacle)
        self.consecutive_visited += 1
    self.prev_pos = current_pos

    # If RoboVac is spending too much time on already visited blocks,
    # start seek mode for a corner or random unvisited block
    if self.consecutive_visited >= 8 and not self.seek_mode:
        self.consecutive_visited = 0
        self.next_unvisited = (-1, -1)
        for corner in random.sample(corners, 4):
            if corner in self.unvisited_blocks:
                self.next_unvisited = corner
        if self.next_unvisited == (-1, -1):
            self.next_unvisited = random.choice(list(self.unvisited_blocks))
        self.seek_mode = True

    ########################################################################
    # Seek mode: where a random unvisited block or corner is selected
    ########################################################################
    while self.seek_mode:
        # If RoboVac is unable to get close enough
        # to the corner or unvisited block with 8 moves,
        # there may be an obstacle in the way;
        # pick another corner or random unvisited block
        if self.consecutive_visited >= 8:
            self.next_unvisited = (-1, -1)
            for corner in random.sample(corners, 4):
                if corner in self.unvisited_blocks:
                    self.next_unvisited = corner
            if self.next_unvisited == (-1, -1):
                self.next_unvisited = random.choice(list(self.unvisited_blocks))
            # Reset consecutive counter, so that algorithm
            # tries different coordinates every 8 moves
            self.consecutive_visited = 0
        print(f"seeking {self.next_unvisited}")
        # Reset priority queue for seek mode
        self.priority_queue = PriorityQueue()
```

```
# Loop through every move in all 4 directions
for dx, dy in directions:
    next_x = current_pos[0] + dx
    next_y = current_pos[1] + dy

    # Check if the next position is within the grid boundaries
    if 0 <= next_x < self.room_width and 0 <= next_y < self.room_height:
        next_pos = (next_x, next_y)

        ########################################################################
        # Heuristic calculations for seek mode
        ########################################################################
        # Check if the next position is not an obstacle
        if next_pos not in self.obstacles:
            # Calculate distance to selected corner or random unvisited block
            # This is a modification of the A* heuristic where
            # previously visited positions are allowed
            priority = self.heuristic2(next_pos, self.next_unvisited)
            self.priority_queue.put(
                (priority, next_pos, directions.index((dx, dy)))
            )

            # Also if an adjacent block that is unvisited
            # (heuristic function validates unvisited)
            # has priority(distance) == 0, move there
            priority = self.heuristic(next_pos)
            if priority == 0:
                self.priority_queue.put(
                    (priority, next_pos, directions.index((dx, dy)))
                )
# Pick lowest distance to unvisited block
if not self.priority_queue.empty():
    priority, next_pos, direction = self.priority_queue.get()
    # Remove the visited block from the set of unvisited blocks
    # Turn off seek mode if RoboVac hits an unvisited block
    if next_pos in self.unvisited_blocks:
        self.unvisited_blocks.remove(next_pos)
        self.consecutive_visited = 0
        self.seek_mode = False
    # Keep counter of consecutive moves to already visited blocks
    else:
        self.consecutive_visited += 1
```

```python
########################################################################
# Loop avoider algorithm
# If the next position has been visited equal or more than 4 times,
# start moving randomly to get out of obstacle
########################################################################
if next_pos not in self.loop_avoider:
    self.loop_avoider[next_pos] = 1
else:
    self.loop_avoider[next_pos] += 1

# Check if the next position has been visited equal or more than 4 times
if self.loop_avoider[next_pos] >= 4:
    while True:
        # pick a direction that doesn't collide with an obstacle
        random_direction = random.choice(directions)
        next_pos = (
            current_pos[0] + random_direction[0],
            current_pos[1] + random_direction[1],
        )
        if next_pos not in self.obstacles:
            print(
                f"loop avoider direction: {directions.index(random_direction)}"
            )
            break
    # If we visit an unvisited block, break out of seek mode
    # and use normal A* search instead
    if next_pos in self.unvisited_blocks:
        self.consecutive_visited = 0
        self.seek_mode = False
    else:
        self.consecutive_visited += 1
    # To avoid loops, we move RoboVac in a certain direction 3 times
    # by adding moves in the same direction 2 times to loop avoider queue
    # and returning that direction 1 time
    for i in range(1, 4):
        next_pos = (
            current_pos[0] + random_direction[0] * i,
            current_pos[1] + random_direction[1] * i,
        )
        if next_pos not in self.obstacles:
            self.loop_avoider_queue.put(
                (0, next_pos, directions.index(random_direction))
            )
            self.loop_avoider[next_pos] = 0
    print("loop avoider activated")
    return directions.index(random_direction)
```

```
        ########################################################################
        # End Loop Avoider Algorithm
        ########################################################################
        # If not loop avoider,
        # move in direction towards corner or random unvisited block
        self.prev_direction = direction
        return direction

########################################################################
# End Seek Mode Code
########################################################################
# Define normal A* search sequence here
########################################################################

# Reset priority queue for regular A* search
self.priority_queue = PriorityQueue()
for dx, dy in directions:
    next_x = current_pos[0] + dx
    next_y = current_pos[1] + dy

    # Check if the next position is within the grid boundaries
    if 0 <= next_x < self.room_width and 0 <= next_y < self.room_height:
        next_pos = (next_x, next_y)

        # Check if the next position is not an obstacle and not visited
        # Calculate the distance(priority) based on the Manhattan distance
        if next_pos not in self.obstacles and next_pos in self.unvisited_blocks:
            priority = self.heuristic(next_pos)
            self.priority_queue.put(
                (priority, next_pos, directions.index((dx, dy)))
            )

# Choose the direction and next position with the lowest distance(priority)
if not self.priority_queue.empty():
    priority, next_pos, direction = self.priority_queue.get()

    # Remove the visited block from the set of unvisited blocks
    if next_pos in self.unvisited_blocks:
        self.unvisited_blocks.remove(next_pos)
        self.consecutive_visited = 0
    else:
        self.consecutive_visited += 1

    self.prev_direction = direction
    return direction

# If no valid move is found, pick random direction to move in
# Random direction must not be in obstacles
```

```python
while True:
    random_direction = directions.index(random.choice(directions))
    next_x = current_pos[0] + directions[random_direction][0]
    next_y = current_pos[1] + directions[random_direction][1]
    if (next_x, next_y) not in self.obstacles:
        break
if (next_x, next_y) in self.unvisited_blocks:
    self.unvisited_blocks.remove((next_x, next_y))
    self.consecutive_visited = 0
else:
    self.consecutive_visited += 1
self.prev_direction = random_direction
print("main random")
return random_direction
```

Obstacles provided code:

```
"""
create robot vacuum that cleans all the floors of a grid.
main creates an instance of RoboVac (your code) and provides:
- grid size
- loc of robovac
- list of x,y,w,h tuples are instance of rectangluar blocks

goal: visit all tiles
exec will : create instance and in game loop call : nextMove()  ??
"""
import random
from queue import PriorityQueue
class RoboVac:
    def __init__(self, config_list):
        self.room_width, self.room_height = config_list[0]
        self.pos = config_list[1]  # starting position of vacuum
        self.block_list = config_list[2]  # blocks list (x,y,width,ht)

        self.obstacles = set()
        self.current_pos = (self.pos[0], self.pos[1])
        self.unvisited_blocks = set()
        # Record coordinates for all blocks(obstacles)
        for block in self.block_list:
            print(block)
            self.add_obstacle(block)
        self.priority_queue = PriorityQueue()
        self.initialize_unvisited_blocks()

        self.loop_avoider = {}

        # fill in with your info
        self.name = "Sanjee Yogeswaran"
        self.id = "47514289"

    def initialize_unvisited_blocks(self):
        # Initialize the set of unvisited blocks
        for x in range(self.room_width):
            for y in range(self.room_height):
                if (x, y) not in self.obstacles:
                    self.unvisited_blocks.add((x, y))


    def add_obstacle(self, obstacle):
        # Add an obstacle to the set of obstacles
        x, y, width, height = obstacle
        for i in range(x, x + width):
            for j in range(y, y + height):
                self.obstacles.add((i, j))


    def heuristic(self, position):
        # Heuristic: Manhattan distance to the closest unvisited block
        min_distance = float("inf")
        for block in self.unvisited_blocks:
            distance = abs(position[0] - block[0]) + abs(position[1] - block[1])
            min_distance = min(min_distance, distance)
        return min_distance


    def get_next_move(self, current_pos):
        # Define possible directions: north, east, south, west
        directions = [(0, -1), (1, 0), (0, 1), (-1, 0)]

        # Create a priority queue for A* search
        priority_queue = PriorityQueue()

        for dx, dy in directions:
            next_x = current_pos[0] + dx
            next_y = current_pos[1] + dy

            # Check if the next position is within the grid boundaries
            if 0 <= next_x <= self.room_width and 0 <= next_y < self.room_height:
                next_pos = (next_x, next_y)

                # Check if the next position is not an obstacle
                if next_pos not in self.obstacles:
                    # Calculate the priority based on the heuristic
                    priority = self.heuristic(next_pos)
                    priority_queue.put((priority, next_pos, directions.index((dx, dy))))

        # Choose the cell with the highest priority (based on the heuristic)
        if not priority_queue.empty():
            _, next_pos, direction = priority_queue.get()
            # Remove the visited block from the set of unvisited blocks
            if next_pos in self.unvisited_blocks:
                self.unvisited_blocks.remove(next_pos)

            if next_pos not in self.loop_avoider:
                self.loop_avoider[next_pos] = 1
            else:
                self.loop_avoider[next_pos] += 1

            if self.loop_avoider[next_pos] >= 4:
                return directions.index(random.choice(directions))
    return direction

# If no valid move is found, stay in the current position (backtrack)
return directions.index(random.choice(directions))
```