Assignment 10

Nakiyah Dhariwala

```
# import libraries
import pandas as pd
import numpy as np
```

Original naive method

```
def cleaning_policy(robot_pos, dirt_map):
    """
    Write your cleaning policy here!

     This  example  is  very  simple.  It  cleans  if  on  dirt,  otherwise  it  moves
randomly.

    Args:
        robot_pos: [row, col] - Current position of the robot
        dirt_map: 2D numpy array where 1 means dirt

    Returns:
        action: Integer 0-4 where:
            0: Move Up
            1: Move Right
            2: Move Down
            3: Move Left
            4: Clean
    """
    # If on dirt, clean it
    if dirt_map[tuple(robot_pos)] == 1:
        return 4

    # Otherwise, move randomly
    return np.random.randint(0, 4)  # Random direction
```

When I ran the original naive base code above, I could evidently see that robot behaved very randomly. If it was in one part of the grid, it continued to stay in that area, moving over the same few cells randomly before moving to another area completely. Even when it moved over to another quadrant/area, it did not follow any pattern/direction. This is also quite evident because the naive code uses np.random.randint(0, 4).

Thus, based on these movements, the robot only cleans when it happens to land on a dirty tile, but it makes no effort to actually look for dirt. Because of this, it wastes a lot of steps moving back and forth over already clean areas. Sometimes it even fails to clean all 20 dirt tiles within the 200-step limit because it gets "stuck" moving randomly instead of exploring the rest of the grid efficiently.

Overall, it was slow, repetitive, and unpredictable.

Section 2: Explain your Solution

2.1 My policy - Greedy Nearest-Dirt Policy with light memory

My improved policy is a nearest-dirt strategy with a little bit of memory. The idea is that at every step, the robot either cleans the tile it's standing on or walks directly toward the closest piece of dirt. In addition, I store the current target in a small global variable so once the robot picks a dirt tile to chase, it sticks with it until it's gone.

This ended up working really well — instead of wandering around like the baseline, the robot always has a goal and reliably cleans all 20 tiles in approximately 51 to 60 steps. My personal best was 51

2.2 Code with comments and a docstring included

```python
# keep track of the current target across calls
_last_target = None


def cleaning_policy(robot_pos, dirt_map):
    """
    Simple nearest-dirt policy with a bit of memory.
    """
    global _last_target
    r, c = robot_pos

    # Clean if we're standing on dirt
    if dirt_map[r, c] == 1:
        _last_target = None  # reset target after cleaning
        return 4

    # Pick a new target if we don't have one or if it's already clean
    if _last_target is None or dirt_map[_last_target] == 0:
        dirt_positions = np.argwhere(dirt_map == 1)

        # nothing left to clean
        if len(dirt_positions) == 0:
            return 0

        # compute Manhattan distance to every dirt tile
        dists = np.abs(dirt_positions[:, 0] - r) + np.abs(dirt_positions[:, 1]
- c)

        # choose the nearest dirt tile
        nearest_idx = np.argmin(dists)
        _last_target = tuple(dirt_positions[nearest_idx])

    # move one step toward the target
    tr, tc = _last_target
    dr, dc = tr - r, tc - c

    # move along whichever direction gets us closer faster
    if abs(dr) >= abs(dc) and dr != 0:
        return 2 if dr > 0 else 0
    elif dc != 0:
        return 1 if dc > 0 else 3
    else:
        # rare case: target disappeared before we reached it
        _last_target = None
        return 4
```

2.3 Discuss your design choices

I wanted something that was simple but still way better than the random baseline. So my code:

- Aims for the nearest dirt: Since the robot can see the whole map, ignoring that information felt wasteful. A nearest-dirt approach means the robot's steps actually count toward the goal.
- Adds a tiny bit of memory: Without remembering the current target, the robot might keep switching between equally close dirt tiles and never actually reach one. Tracking _last_target fixes that so that the robot commits to a choice until it cleans it.
- Uses Manhattan Distance: Since the robot can only move up, down, left, and right, Manhattan distance matches the true movement cost. After trying a few variations, I also kept the movement rule simple by moving along whichever axis has the larger distance so the robot doesn't zig-zag or take odd paths.

Overall, I chose this design because it's simple, easy to interpret, and directly matches the goal of cleaning dirt quickly. Using Manhattan distance makes sense on a grid, and the memory variable keeps the robot from making unstable decisions.

Performance Analysis

3.1: Compare your policy to the baseline

Compared to the baseline using the naive approach, this policy is a lot more efficient. Instead of randomly wandering around, it finishes in around 55-56 steps on average, below the 200-step limit. Even though my approach isn't perfectly optimal and lags bhind the top scorers, it consistently outperforms the random policy by a huge gap.

3.2: Analyze Failure Cases

One issue with my policy is that it isn't always fully thorough in one area before moving on. Sometimes the robot cleans most of the dirt in a quadrant but accidentally leaves behind one or two spots. Because it always goes to the nearest dirt, it might see something slightly closer in another quadrant and head there instead. Later, it realizes it missed those earlier tiles and has to walk back to clean them. This back-and-forth doesn't break the policy, but it does make it a little less efficient.

3.3: Suggest further improvements

Because of the failure case mentioned above, one improvement I would want is to make the policy a bit more aware of clusters of dirt, or to introduce some kind of quadrant-based sweeping. Right now, the robot always picks the single nearest dirt tile, which works well most of the time but can cause it to bounce between areas.

Instead if the robot could "efficiently" look for clusters of dirt tiles that are close to each other and treat them as a group, it could stay in that area and clean all the nearby dirt before moving to a different part of the grid. Similarly, I think a simple quadrant-sweeping rule could help the robot finish an entire section of the grid rather than leaving behind stray tiles. Both of these changes would make its path more consistent and reduce the extra steps caused by backtracking.

However, when I tried adding cluster scoring or quadrant rules, the robot ended up skipping closer dirt and making less efficient choices. The extra logic actually made things worse by creating longer routes and unnecessary backtracking. So even though the ideas make sense, my attempts hurt performance, which is why I stuck with the simpler nearest-dirt approach for now. Nonetheless I do think this should (if implemented correctly) optimize the cleaning.