CSC Problem Set 01 +X Report

Name: John Cheek
Email: jwcheek2@ncsu.edu
Section: 001
Collaborators: None

Summary:

For my +X implementation I decided to play around with a randomized approach to the agent's movement if there were no adjacent dirty tiles available. I created three variations of a randomized algorithm: true random, improved random, and hybrid random. To evaluate these, I modified the test case to run the initial test 100 times on maps 2 and 3 (as the large amount of connected dirty tiles on maps 1, 4, and 5 allowed all the random algorithms to pass consistently on those maps) and report the number of successful tests. While I was ultimately unable to create an algorithm that used randomization and *consistently* solved the problem set, I observed some interesting behavior across the three algorithms.

The best performing algorithm was the 'hybrid randomization' which performed the best on map 2 and perfectly on map 3. The worst performing algorithm was (unsurprisingly) the true random algorithm.

"+X" Concept:

When solving the problem set, the robot would occasionally get 'stuck' when surrounded by a combination of clean or impassable tiles and pass 0 trials. To solve this, I first tried a random algorithm. This approach worked on maps 1, 4, and 5 and passed a non-zero amount of trials on maps 2 and 3 (though consistently far less than 70 trials) which was better than my initial solution, but far from perfect. I ultimately scrapped the random approach because in order to get full credit for the problem set, I figured I should create a consistent solution, but I thought I could play around with a randomized approach and try to improve it for my +X percent on this problem set. Ultimately I tried three different approaches to randomized algorithms which I call true random, improved random, and hybrid random. Each approach begins with prioritizing and moving towards adjacent dirty tiles and all use the Java random library. [1] True random then randomly selects an action, improved random still randomly selects an action but prevents the agent from undoing the random action it just took, and hybrid random combines by finial initial solution of moving towards unvisited tiles with the improved random approach.

Technical Implementation of +X:

Each of the three algorithms I tested start by moving towards adjacent dirty tiles with a simple conditional check. If there are no adjacent dirty tiles it will then act with some aspect of randomness depending on the algorithm.

```
// Check for dirty tiles and move towards it (Up, Right, Down, Left)
if ( above != null && above.getStatus() == TileStatus.DIRTY ) {
    previousAction = Action.MOVE_UP;
    return Action.MOVE_UP;
}
if ( right != null && right.getStatus() == TileStatus.DIRTY ) {
    previousAction = Action.MOVE_RIGHT;
    return Action.MOVE_RIGHT;
}
if ( below != null && below.getStatus() == TileStatus.DIRTY ) {
    previousAction = Action.MOVE_DOWN;
    return Action.MOVE_DOWN;
}
if ( left != null && left.getStatus() == TileStatus.DIRTY ) {
    previousAction = Action.MOVE_LEFT;
    return Action.MOVE_LEFT;
}
```

The agent's behavior will then deviate based on some aspect of randomness in the following three random movement algorithms

True Random:

For the true random algorithm a number between 0 and 3 is randomly selected and it simply moves according to that result. 0 representing ‘move up’, 1 for ‘move right’, 2 for ‘move down’, and 3 for ‘move left’:

```
final Random rand = new Random();
final int direction = rand.nextInt( 4 );

switch ( direction ) {
    case 0:
        if ( above != null && above.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_UP;
            return Action.MOVE_UP;
        }
        break;
    case 1:
        if ( right != null && right.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_RIGHT;
            return Action.MOVE_RIGHT;
        }
        break;
    case 2:
        if ( below != null && below.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_DOWN;
            return Action.MOVE_DOWN;
        }
        break;
    case 3:
        if ( left != null && left.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_LEFT;
            return Action.MOVE_LEFT;
        }
        break;
    default:

}
```

Improved Random:

The improved random algorithm acts similarly however, it will avoid undoing the previous move action. This is done by keeping track of the previous movement action and preventing it from taking the opposite action if that action is randomly chosen. For instance, if in one step the robot chooses ‘move left’ and in the next step ‘move right’ is randomly chosen, it will not move right (as it will just be undoing the previous action it took) and will instead default to moving away if possible (in this case, move to the left again). The algorithm utilizes conditional statements and the previousAction field to determine its movement.

```
final Random rand = new Random();
final int direction = rand.nextInt( 4 );

switch ( direction ) {
    case 0:
        if ( previousAction == Action.MOVE_DOWN && above != null
                && above.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_UP;
            return Action.MOVE_UP;
        }
        else if ( previousAction == Action.MOVE_UP && below != null
                && below.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_DOWN;
            return Action.MOVE_DOWN;
        }
        break;
    case 1:
        if ( previousAction != Action.MOVE_LEFT && right != null
                && right.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_RIGHT;
            return Action.MOVE_RIGHT;
        }
        else if ( previousAction == Action.MOVE_RIGHT && left != null
                && left.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_LEFT;
            return Action.MOVE_LEFT;

        }
        break;
    case 2:
        if ( previousAction != Action.MOVE_UP && below != null && below.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_DOWN;
            return Action.MOVE_DOWN;
        }
        else if ( previousAction == Action.MOVE_UP && above != null
                && above.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_UP;
            return Action.MOVE_UP;

        }
        break;
    case 3:
        if ( previousAction != Action.MOVE_RIGHT && left != null
                && left.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_LEFT;
            return Action.MOVE_LEFT;
        }
        else if ( previousAction == Action.MOVE_LEFT && right != null
                && right.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_RIGHT;
            return Action.MOVE_RIGHT;

        }
        break;
    default:

}
```

Hybrid Random:

The hybrid random algorithm is a modified version of my original solution and utilizes a set to keep track of visited tiles. It starts just like my original solution where it first checks for dirty tiles, then checks for adjacent unvisited tiles, but then deviates if no dirty or unvisited tiles are adjacent. Instead of utilizing a stack to keep track of previous actions and backtrack if the robot gets “stuck,” it will move randomly utilizing the improved random logic until adjacent to a dirty or unvisited tile. I considered modifying the randomization logic to prevent moving to visited tiles, but that would potentially create a scenario where the robot gets stuck when surrounded by visited tiles which would be challenging to solve using only randomization.

```
if ( above != null && !visitedPositions.contains( above ) && above.getStatus() != TileStatus.IMPASSABLE ) {
    previousAction = Action.MOVE_UP;
    return Action.MOVE_UP;
}
if ( right != null && !visitedPositions.contains( right ) && right.getStatus() != TileStatus.IMPASSABLE ) {
    previousAction = Action.MOVE_RIGHT;
    return Action.MOVE_RIGHT;
}
if ( below != null && !visitedPositions.contains( below ) && below.getStatus() != TileStatus.IMPASSABLE ) {
    previousAction = Action.MOVE_DOWN;
    return Action.MOVE_DOWN;
}
if ( left != null && !visitedPositions.contains( left ) && left.getStatus() != TileStatus.IMPASSABLE ) {
    previousAction = Action.MOVE_LEFT;
    return Action.MOVE_LEFT;
}

final Random rand = new Random();
final int direction = rand.nextInt( 4 );

switch ( direction ) {
    case 0:
        if ( previousAction != Action.MOVE_DOWN && above != null
                && above.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_UP;
            return Action.MOVE_UP;
        }
        else if ( previousAction == Action.MOVE_UP && below != null
                && below.getStatus() != TileStatus.IMPASSABLE ) {
            previousAction = Action.MOVE_DOWN;
            return Action.MOVE_DOWN;

        }
        break;
    case 1:
```

This code shows the additional logic from my original solution which is added before the random case (the code for the random case is identical to the Improved Random algorithm)

Evaluation and Results:

To evaluate my results, I modified the original test case to repeat 100 times on Maps 2 and 3, keeping track of the amount of times the original test would have passed. I then ran each algorithm ten times and recorded the results from each test as the number of times the original test case passed, then calculated the average completion percentage for each algorithm across the 10 trials.

Here are the full results (for fun):

	Trial:	1	2	3	4	5	6	7	8	9	10
Map 2:	True Random	0	0	0	1	0	0	0	0	0	0
	Improved Random	6	12	5	10	7	6	6	5	5	2
	Hybrid Random	11	11	5	6	10	5	6	11	7	9
Map 3:	True Random	0	0	0	0	0	0	0	0	0	0
	Improved Random	0	0	0	0	0	0	0	0	0	0
	Hybrid Random	100	100	100	100	100	100	100	100	100	100

And then the statistics for each:

Map 02:

Algorithm	Average Tests Passed (%)	Max Tests Passed	Min Tests Passed
True Random	0.1	1	0
Improved Random	6.4	12	2
Hybrid Random	8.1	11	5

Map 03:

Algorithm	Average Tests Passed (%)	Max Tests Passed	Min Tests Passed
True Random	0	0	0
Improved Random	0	0	0
Hybrid Random	100	100	100

On map two, the algorithms performed as expected; the more complicated the randomization logic, the better on average they performed. This is likely because of the scattered placement of the impassable tiles on the map. Moving randomly around them would not solve the problem as

random movements can drive the agent further away from dirty tiles, but as more support for avoiding random movement is built into the algorithm, the better it performs.

On map three, I was surprised with how neither the true nor improved random algorithm did not have a single success. Upon running the visualization multiple times it was likely because the random movement made it difficult to get through the 2x2 tunnel. between the impassable tiles in the map. It cleans the left half of the map consistently but since all of those tiles are now clean it moves randomly all across that half of the map, rarely reaching the 2x2 tunnel. The Hybrid random solves this by prioritizing unvisited tiles which will lead it to the tunnel and then be adjacent to a large amount of dirty tiles on the right hand side.

Works Referenced:

1. *Oracle, “Random (Java Platform SE 8),” Oracle.com, Jan. 06, 2020. https://docs.oracle.com/javase/8/docs/api/java/util/Random.html*