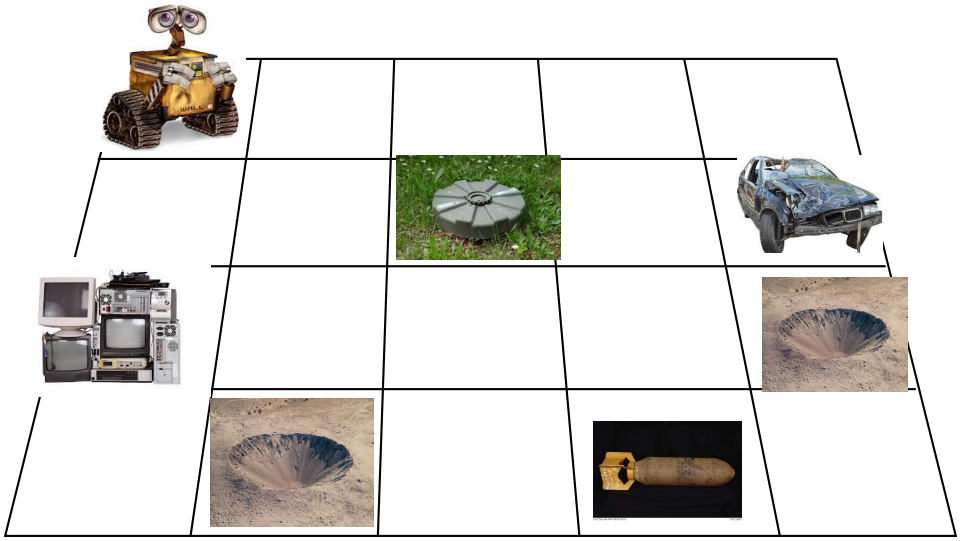
Project D: WALL-E

Due: Tuesday 4/12/2022, 11:59pm ET

1 Background

It's the year of 2805, 100 years after human being deserted the earth after a nuclear war that left the once beautiful planet completely inhabitable. WALL-E, a robot left behind in the ruins is still working as he was initially programmed. Every day he sets out to collect recyclable materials for his own maintenance and to provide energy to rejuvenate the planet. This work is not easy as he has to navigate around craters and undetonated explosives. WALL-E is driven by battery. Moving into a crater will cost him extra battery energy, and moving into an explosive will blown him up into oblivion. Your job is to develop a program to help him efficiently and safely collect all the recyclables and clear all the explosives in a field before his battery runs out.



2 Project Specifications

WALL-E works on a field of dimensions `Lx` × `Ly`, which is divided by `Lx` × `Ly` unit square grids. Each grid has coordinates (x, y), where x = 0, 1, ..., Lx-1, and y = 0, 1, ..., Ly-1. There are four types of grid, represented by different integer numbers, 0 (clean space), 1 (recyclable material), 2 (undetonated explosive), or 3 (crater). Grid (0,0), where WALL-E starts, is always a clean space.

WALL-E is equipped with optical sensors to detect the 4 grids immediately adjacent to him. The optical sensors will return a list of 4 integers representing the type of the grid in the north, east, south, and west directions, respectively. If he is already at the boundary grid of the field, the sensor corresponding to that direction will return -1. For example, sensor signals [0, 1, 2, -1] means that the grid immediate to the north is a clean space, the grid immediate to the east

has a recyclable material, the grid immediate to the south has an explosive, and he cannot move toward west since he already reaches the west boundary of the field.

He is moving around the field, and the battery consumption depends on the grid he moves into. Moving into a clean space consumes 1 battery point, and moving into a crater consumes 5 battery points. He uses his sensors all the time. Whenever he detects a recyclable material and/or a undetonated explosive in any grid immediately adjacent to him, he will stay at that location and collect the materials and/or clear the explosives. Collecting recyclable and clearing explosive do NOT consume battery points. After cleaning the recylclables/explosives, those grids become clean spaces, and he keeps on moving. For the first generation design, the moving direction is random.

During the war, WALL-E received modernization and was upgraded to the second generation with many new features, including the long-range magnetic field sensor. The long-range magnetic field sensor can detect the magnetic field of the recyclable materials and undetonated explosives from far away. Each of the recyclable/explosive generates a magnetic field that gradually decays with the distance. The magnetic field strength follows the inverse cube law, i.e., the magnetic field strength at distance d from its source is $1/d^3$ for $d > 0$, that is, the magnetic field at the source location is zero. For example, a recyclable material at coordinate (1,1) will generate a magnetic field of strength 0 at (1,1), and strength 0.008 at (4,5). Let's assume that the magnetic field strengths of all the recyclable materials and the undetonated explosives are the same. Then at any location in the field, the total magnetic field strength is $\sum_i 1/d_i^3$, where i loops over all the recyclable materials and the undetonated explosives. The magnetic field sensor, along with many other new features, significantly improve the WALLE's work efficiency.

3 Programming Tasks (Total 100 points)

3.1 Developing environment

Use Anaconda Spyder as your developing environment. To show animation properly, go to: Tools > Preferences > IPython Console > Graphics > Backend and change it from "Inline" to "Automatic".

3.2 Template files

You are provided with a template that includes four python scripts, `utils.py`, `field.py`↩, `robot.py`, and `main.py`. Some functionalities are already implemented. Understanding them can help you complete the project. You need to complete all the `TODO` sections.

The template also includes three files `testmap_sol.txt`, `demo_sol.txt`, and `demo_sol.gif` for testing and debugging purposes. Do NOT modify these files. They are to be compared with the files `testmap.txt`, `demo.txt`, and `demo.gif` you generated while running your program. **Place all the scripts and testing files under the same folder**.

3.3 Complete functions in `utils.py`

3.3.1 `generate_2D_map()`

Function `generate_2D_map(Lx, Ly, nRecyclable, nExplosive, nCraters)` is given. It generate a 2D list representing a 2D map of dimensions `Lx` × `Ly`, with the given numbers of recyclable materials, undetonated explosives, and craters, the rest being clean spaces. The values of the elements of the list are 0 (clean space), 1 (recyclable), 2 (explosive), or 3 (creater). Recyclables, explosives, and craters are randomly distributed in the map, except that the (0,0) grid is always a clean space because that is where WALL-E starts. The dimensions are chosen so that the first level of the 2D list has `Lx` elements, where each elements is a 1D list of `Ly` elements. For example, `generate_2D_map(4, 3, 1, 1, 1)` should return `[[0, 0, 0], [0, 0, 1], [2, 0, 0], [0, 0, 3]]`.

You might wonder why this supposedly random map generator produces a predictable map. This is because the random numbers generated in computer programs are not real random numbers, but pseudo random numbers that mimic real random numbers. Pseudo random numbers are generated from a seed. Same seed always generates the same sequence of pseudo random numbers. A common practice is to use system time in milliseconds as a seed value. In the `generate_2D_map()` function, we fix the seed using `random.seed(value)` statement to generate same sequence of pseudo random numbers repeatedly for testing and debugging.

3.3.2 `write_2D_to_txt()` (4 points)

Complete function `write_2D_to_txt(map2D, filename)`, which writes a 2D list, `map2D`, into a file with the given filename. The file should be written in a way that (0,0) grid is the top left (North West) corner, and the x coordinate increases to the right (East) and the y coordinate increases to the bottom (South). In other words, x is the width direction and y is the height direction. For example, if the 2D list `map2D = [[0, 1], [1, 2], [0, 3]]`, then `write_2D_to_txt(map2D, 'testmap.txt')` should generate a file `testmap.txt` that reads

```
0 1 0
1 2 3
```

Note: in the txt file there is one space between adjacent numbers on the same line, but there is no space at the end of the line.

3.3.3 `read_2D_from_txt()` (4 points)

Complete function `read_2D_from_txt(filename)`, which reads a map from a file with the given filename into a 2D list. This function is the inverse of `write_2D_to_txt()`. For example, with the `testmap.txt` generated from above, `read_2D_from_txt('testmap.txt')` should return a 2D list `[[0, 1], [1, 2], [0, 3]]`.

3.3.4 `print_2D()`

Function `print_2D(map2D)` is given. It is similar to `write_2D_to_txt(map2D, filename)`. Instead of writing to a file, it prints the 2D list on the screen, using x as the width direction, and y as the height direction. This function is for testing and debugging purposes.

3.3.5 Test your functions in `utils.py`

A number of test functions are provided in `utils.py`. After you complete the tasks above, you can run `utils.py` to test your implementation.

3.4 Complete `field` class in `field.py`

3.4.1 `__init__()` (6 points)

The constructor `__init__(self, fieldmap)` takes a 2D list, `fieldmap`, that represents a field map as the input variable. The field map can be generated using the `generate_2D_map()` or the `read_2D_from_txt()` functions. The `field` class has the following member attributes (1 point for each attribute):

`Lx`: integer number, the width of the field.
`Ly`: integer number, the height of the field.
`map`: 2D list of integer numbers, storing the field map.
`magMap`: 2D list of floating numbers, the magnetic field at each grid in the field.
`nRecyclable`: integer number, number of recyclable materials remained in the field.
`nExplosive`: integer number, number of undetonated explosives remained in the field.

Note: Attributes `magMap`, `nRecyclable`, and `nExplosive` should be initialized by calling the corresponding methods defined below.

3.4.2 Methods in `field` class (14 points)

The `field` class has the following methods:

`calcNumRecyclable(self)`: calculate the number of recyclables remained in the field, and pass the value to member attribute `nRecyclable`. (2 points)

`calcNumExplosive(self)`: calculate the number of explosives remained in the field, and pass the value to member attribute `nExplosive`. (2 points)

`calcMagMap(self)`: calculate the magnetic field for all grids, as a 2D list, and pass the results to member attribute `magMap`. See Sec.2 for how to calculate the magnetic field. (5 points)

`getGrid(self, x, y)`: return the value of grid (grid type) at coordinates `(x, y)`. (1 point)

`setGrid(self, x, y, value)`: set the value of grid (grid type) at coordinates `(x, y)` to the given `value`. (1 point)

`getMagGrid(self, x, y)`: return the magnetic field at the grid at coordinates `(x, y)`. (1 point)

`update(self)`: recalculate the member attributes `nRecyclable`, `nExplosive`, and `magMap` due to the field map change. (2 points)

3.4.3 Test your implementation of the `field` class

After you complete the tasks above, use the `if __name__=="__main__"` block to test your implementation. Some examples are already provided. Running `field.py` with the provided examples should print out the following to the console

```
0 0 0 0
0 0 0 0
0 0 0 1
4 3
0
1 0 0 0
0 0 0 0
0 0 0 1
```

3.5 Complete `robot` class in `robot.py`

The `robot` class represents the first generation design. The robot relies on optical sensors to detect objects. The direction of robot's motion is random.

3.5.1 `__init__()` (12 points)

The constructor `__init__(self, x, y, batt)` takes coordinates `(x, y)` as the initial position of the robot, which is always `(0, 0)`, and `batt` as the initial battery level. The `robot` class has the following member attributes (1 point for each attribute):

`version`: interger number 1, representing first generation.

`x`: integer number, the x coordinate of the robot's current position.

`y`: integer number, the y coordinate of the robot's current position.

`batt`: integer number, the current battery level.

`sensors`: list of 4 integer numbers representing optical sensor signals from immediately adjacent grids in the north, east, south, and west direction, respectively.

`positions`: list of (x, y) tuples as the history of robot position.

`battlevels`: list of integers as the history of battery point.

`actionLists`: list of strings that represent all possible actions a robot can take, ["north", "east", "south", "west", "clean"]. All the strings are lowercase.

`actions`: list of strings as the history of robot action. Action can be any one item in `actionList`.

`nRecyclable`: integer number, number of recyclable materials collected by the robot. Initial value 0.

`nExplosive`: integer number, number of undetonated explosives cleared by the robot. Initial value 0.

`nextMove`: integer number, representing possible moves: 0 (north), 1 (east), 2 (south), 3 (west), 4 (clean). Initial value -1.

Note: `sensors`, `positions`, `battlevels`, and `actions` should be initialized as empty lists.

3.5.2 Methods in `robot` class (32 points)

The `robot` class has the following methods:

`reportStatus(self)`: save current position and battery points to the history, member attributes `positions` and `battlevels`, respectively. (2 points)

`readSensors(self, field)`: use the optical sensors to probe the 4 immediately adjacent grids in the field, and pass the signals to member attributes `sensors`. The `sensors` attribute is a list of 4 integers. See Sec.2 for details. (4 points)

`selectNextMove(self, field)`: read the optical sensors, and based on the member attributes `sensors`, select the next move, represented by an integer, 0 (move north), 1 (move east), 2 (move south), 3 (move west), 4 (clean). Save this integer to member attribute `nextMove`. If the sensors indicate that there is a recyclable/explosive nearby, 4 has to be chosen. Otherwise, the moving direction is chosen in random. However, it cannot choose a move that will get the robot out of the boundaries. (6 points)

`setNextMove(self, strNextMove)`: based on input `strNextMove`, set the member attribute `nextMove`. When `strNextMove` takes value "north", "east", "south", "west", or "clean", member attribute `nextMove` takes value 0, 1, 2, 3, or 4, respectively. This method is mostly for debugging purpose. (2 points)

`action(self, field)`: based on the value of member attribute `nextMove`, call respective method defined below. For `nextMove` value 0, 1, 2, 3, or 4, call method `moveNorth()`, `moveEast()`, `moveSouth()`, `moveWest()`, or `collect_n_clear()`, respectively. Append action to history, member attribute `actions`. (2 points)

`moveNorth(self, field)`: check if the robot is at the north boundary, if not then move the robot one grid north (-y direction), otherwise do nothing. Calculate battery consumption and update the member attribute `batt`. Moving into a crater costs 5 battery points, and moving into a clean space costs only 1 battery point. (2 points)

`moveEast(self, field)`: check if the robot is at the east boundary, if not then move the robot one grid east (+x direction), otherwise do nothing. Calculate battery consumption and update the member attribute `batt`. Moving into a crater costs 5 battery points, and moving into a clean space costs only 1 battery point. (2 points)

`moveSouth(self, field)`: check if the robot is at the south boundary, if not then move the robot one grid north (+y direction), otherwise do nothing. Calculate battery consumption

and update the member attribute `batt`. Moving into a crater costs 5 battery points, and moving into a clean space costs only 1 battery point. (2 points)

`moveWest(self, field)`: check if the robot is at the west boundary, if not then move the robot one grid west (-x direction), otherwise do nothing. Calculate battery consumption and update the member attribute `batt`. Moving into a crater costs 5 battery points, and moving into a clean space costs only 1 battery point. (2 points)

`collect_n_clear(self, field)`: collect recyclables and/or clear explosives in four grids immediately adjacent to the robot, and change the respective grids to clean spaces. Keep track of the member attributes `nRecyclable` and `nExplosive`. Call `update()` method in the `field` class to update the magnetic field and object counts. (8 points)

`workLog(self, filename)`: this method is given. It outputs the working history to a file.

Note: If the battery is positive before action `moveNorth()`, `moveEast()`, `moveSouth()`, or `moveWest()`, we assume that the action will complete even if battery drops below zero during the action. For example, when the battery is 2 before moving into a crater, the action can still be completed with remaining batter -1. However, if the battery is 0, then no collecting/clearing action will happen even though they do not cost battery.

3.5.3 Test your implementation of the `robot` class

After you complete the tasks above, use the `if __name__=="__main__"` block to test your implementation. Some examples are already provided. Running `robot.py` with the provided examples should print out the following to the console

```
0 0 0 0
0 0 0 0
0 0 0 1
[-1, 0, 0, -1]
0 0 10
0 0 10
```

3.5.4 Simulate and visualize the first generation WALL-E

After you complete the tasks above, run `main.py` to perform and visualize a simulation of the first generation WALL-E. Choose the `'demo'` mode in the `TODO` section in `main.py`, where you can also customize the field, the robot and the animation settings. If needed, study the `simulate()` function in `main.py` to get a better understanding of how WALL-E works.

3.6 Complete `robotGen2` subclass in `robot.py`

WALL-E received modernization during the war. The `robotGen2` class represents this second generation design. It inherits the `robot` class. It has many new features, including the long-range magnetic field sensor to improve the work efficiency.

3.6.1 `__init__()` (3 points)

This subclass has the following member attributes (1 point for each attribute):

`version`: interger number 2, representing second generation.

`magField`: floating number, magnetic field at current position. Initial value 0.

`prev_magField`: floating number, magnetic field before the last action. Note: last action might or might not change robot position. Initial value 0.

3.6.2 Methods in `robotGen2` subclass (15 points)

This subclass has the following methods:

`readMagSensor(self, field)`: get the magnetic field strength at the current position, and pass the value to member attribute `magField`. Need to take care of member attribute `prev_magField` as well. (2 points)

`selectNextMove(self)`: this method should be overloaded to use both the optical sensors and the magnetic sensor to make more efficient moves. (1 point for correctly implementing a design different from the first generation, 12 points for better performance in all cases during final evaluation, see Sec.3.7)

3.6.3 Additiona features

You are welcome to add more member attributes and methods in `robotGen2` class to further improve WALL-E's work efficiency. Use your imaginations. Sky is the limit.

3.6.4 Test your implementation of the `robotGen2` class

After you complete the tasks above, use the `if __name__=="__main__"` block to test your implementation.

3.6.5 Simulate and visualize the second generation WALL-E

After you complete the tasks above, run `main.py` to perform and visualize a simulation of the second generation WALL-E. Choose the `'demo'` mode in the `TODO` section in `main.py`. Customize the robot to be second generation.

3.7 Final evaluation and performance comparison

Run `main.py`, choosing the `'final'` mode in the `TODO` section. Compare the performance of the first and the second generations design. WALL-E will start at (0,0) with 500 battery points. The evaluation will be based on 5 given fields:

10×10 field with 10 recyclables, 10 explosives, and 10 craters
15×15 field with 15 recyclables, 15 explosives, and 15 craters
20×20 field with 20 recyclables, 20 explosives, and 20 craters

25×25 field with 25 recyclables, 25 explosives, and 25 craters
30×30 field with 30 recyclables, 30 explosives, and 30 craters

The performance is measured by the numbers of recyclables/explosives cleaned. If all recyclables/explosives are cleaned, the performance is measured by the battery points left.

Note: do NOT hardcode your moves! We will used hidden maps to test your implementation.

3.8 Document the design and performance (5 points)

In a document, `report.pdf`, explain your second generation design in details. Use a table to compare the performance of the first generation and the second generation, based on the final evaluation results. Report the remaining battery points, number of recyclable collected and number of explosives cleared in each case. You are welcome to include additional materials, but keep your document within 5 pages.

3.9 Coding style (5 points)

Code in good style. Add docstrings for all classes and functions. Provide comments in the code whenever necessary to make your code more understandable. Refrain from using long lines. Use spaces, line breaks, and # symbols properly to improve the readability of your code. See `http://peps.python.org/pep-0008/` for complete style guide for Python code.

4 Submission

Please submit `utils.py`, `field.py`, `robot.py`, `main.py`, and `report.pdf` to GradeScope.