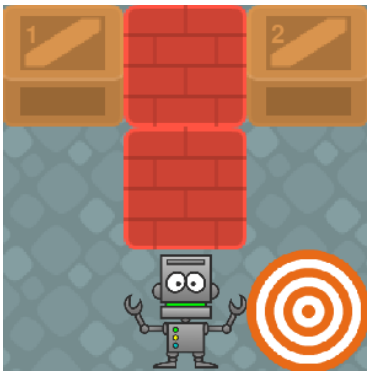
CS 7638: Artificial Intelligence for Robotics

Warehouse Project - Fall 2021 - Deadline: Monday November 15th, Midnight AOE



Introduction

In this project, you will develop algorithms to guide a robot through a warehouse to pick up and deliver boxes to a designated drop zone area. The template code provides 3 classes, one for each part of the project: `DeliveryPlanner_Part[A, B, C]`

- You may share code between part A, B, and C
- Your submission will consist of a single file: `warehouse.py`
- Your `warehouse.py` file must not execute any code when imported

Grading

- The weighting for each part is:
  - Part A = 40%
  - Part B = 40%
  - Part C = 20%
- Within each part, each test case is equally weighted.
- Part A and B have opportunities for extra credit. Any extra credit earned in these parts will supplement lost points in another part. The overall max score for this project is 101% (1% extra credit).

Part A (40%)

Your task is to pick up and deliver all boxes listed in the `todo` list. You will do this by providing a list of actions that the testing suite will execute in order to complete all the deliveries. Your algorithm should determine the best path to take when completing this task.

`DeliveryPlanner_PartA's` constructor must take three arguments: `self`, `warehouse`, and `todo`. It must also have a method called `plan_delivery` that takes 2 arguments: `self` and `debug` (set to `False` by default).

Part A Input Specifications

`warehouse` will be a list of `m` strings, each with `n` characters, corresponding to the layout of the warehouse. The warehouse is an `m` x `n` grid. `warehouse[i][j]` corresponds to the spot in the `i`th row and `j`th column of the warehouse, where the `0`th row is the northern end of the warehouse and the `0`th column is the western end.

The characters in each string will be one of the following:
`.` (period) : traversable space. The robot may enter from any adjacent space.

`#` (hash) : a wall. The robot cannot enter this space.

`@` (dropzone) : the starting point for the robot and the space where all boxes must be delivered. The dropzone may be traversed like a `.` space.

`[0-9a-zA-Z]` (any alphanumeric character) : a box. At most one of each alphanumeric character will be present in the warehouse (meaning there will be at most 62 boxes). A box may not be traversed, but if the robot is adjacent to the box, the robot can pick up the box. Once the box has been lifted from a space, the space functions as a `.` (period).

Note: Test cases in the test suite will only contain the characters listed above. There is a helper function (`_set_initial_state_from`) that parses this initial input into an internal warehouse state. This is the same internal state representation that is used by the testing suite. You are not required to use this helper function, it is just provided for convenience. The helper function includes one additional character denoting the location of the robot:

`*` (asterisk) : the current location of the robot. When the current location of the robot is the same as the dropzone then the warehouse cell will be `*` instead of `@`.

For example,

```
warehouse = ['1#2',
             '.#.',
             '..@']
```

is a 3x3 warehouse.

- The dropzone is at the warehouse cell in row 2, column 2.
- Box `1` is located in the warehouse cell in row 0, column 0.
- Box `2` is located in the warehouse cell in row 0, column 2.
- There are walls in the warehouse cells in row 0, column 1 and row 1, column 1.
- The remaining five warehouse cells (which includes the dropzone) are traversable spaces.
- After this warehouse is parsed using the helper function (`_set_initial_state_from`), the `@` will be replaced with a `*` because the robot's starting location is the dropzone. The testing suite has its own copy of the warehouse state used to execute your delivery plan to check for success. In the testing suite, after the robot moves out of the dropzone cell, the dropzone is denoted with the `@` again. You are free to continue with this convention in your code, or choose another convention of keeping track of the robot, dropzone, walls, and boxes. Either way, you must update your internal warehouse state in your code as this is not done for you.

The argument `todo` is a list of alphanumeric characters giving the order in which the boxes must be delivered to the dropzone. For example, if `todo = ['1','2']` is given with the above example warehouse, then the robot must first deliver box `1` to the dropzone, and then the robot must deliver box `2` to the dropzone.

Part A Rules & Costs for Actions

- The robot may move in 8 direction (N, E, S, W, NE, NW, SE, SW)
- The robot may not move outside the warehouse. The warehouse does not "wrap" around (it is not cyclic).
- Two spaces are considered adjacent if they share an edge or a corner.
- The robot may pick up a box that is in an adjacent square.
- The robot may put a box down in an adjacent square, so long as the adjacent square is empty (`.` or `@`).
- While holding a box, the robot may not pick up another box.
- The robot may move horizontally or vertically at a cost of 2 per move.
- The robot may move diagonally at a cost of 3 per move.
- The cost to pick up a box is 4 (and 2 to put down) (regardless the direction).
- If a box is placed on the `@` space, it is considered delivered and is removed from the warehouse, thus the `@` space is still traversable after dropping a box on it.

- The warehouse will be arranged so that it is always possible for the robot to move to the next box on the todo list without having to rearrange any other boxes.
- The robot will stay in the same location when an illegal move is executed.
- An illegal move will incur a cost of 100 in addition to the action cost.
- Illegal moves include:
  - attempting to move to a nonadjacent, nonexistent, or occupied space
  - attempting to pick up a nonadjacent or nonexistent box
  - attempting to pick up a box while holding one already
  - attempting to put down a box on a nonadjacent, nonexistent, or occupied space (this means the robot may not drop a box on the drop zone while the robot is occupying the drop zone)
  - attempting to put down a box while not holding one

Part A Method Return Specifications

`plan_delivery` should return a `list` of moves that minimizes the total cost of completing the task. Each move should be a string formatted as follows:

- `'move {d}'`, where `'{d}'` is replaced by the direction the robot should move: `"n", "e", "s", "w", "ne", "se", "nw", "sw"`
- `'lift {x}'`, where `'{x}'` is replaced by the alphanumeric character of the box being picked up
- `'down {d}'`, where `'{d}'` is replaced by the direction the robot will put the box down

For example, for the values of `warehouse` and `todo` given previously (reproduced below):

```
warehouse = ['1#2',
             '.#.',
             '..@']

todo = ['1','2']
```

`plan_delivery` might return the following:

```
['move w',
 'move nw',
 'lift 1',
 'move se',
 'down e',
 'move ne',
 'lift 2',
 'down s']
```

Part A Scoring

The testing suite will execute your plan and calculate the total cost: `student_cost`. The score for each test case will be calculated by: `benchmark_cost` / `student_cost`. The benchmark will be greater than or equal to the absolute minimum cost, which leaves an opportunity for extra credit on some test cases. You will receive a 0 in the following siturations:

- your code takes longer than the prescribed time limit
- your method returns the wrong output format
- the boxes are not delivered in the correct order

Part B (40%)

In this part there are three main differences from part A:

- there will be only a single box for your robot to deliver
- the warehouse has an "uneven" floor which imposes an additional, non-negative, cost on robot actions

- the robot starting location is not provided

`DeliveryPlanner_PartB's` constructor must have four arguments: `self`, `warehouse`, `warehouse_cost`, and `todo`. It must also have a method called `plan_delivery` that takes two arguments: `self` and `debug` set to `False` by default.

Part B Input Specifications

Same as part A but the only box in the warehouse will be: `1` (the single box to be delivered).

For example:

```
warehouse = ['1..',
             '.#.',
             '..@']
```

The argument `warehouse_cost` is a `list` of `lists` such that indices `i,j` refer to the cost at the row `i` and column `j` in the warehouse. For the case above, the corresponding warehouse_cost could be:

```
warehouse_cost = [[ 0,        5, 2],
                  [10, infinity, 2],
                  [ 2,       10, 2]]
```

where the interior wall has a cost of infinity. The maximum value for a non-wall cell in warehouse_cost will be 100.

The argument `todo` is limited to a single box as follows:
`todo = ['1']`

There is no input for initial robot location because the robot may "wake up" at any point in the warehouse and must be handed a "policy" so that no matter where it is, it can retrieve the box. Further, because it may lift the box from different squares depending on its starting location, it requires another "policy" to deliver the box to the dropzone.

Part B Rules for Actions

Same as part A.

Part B Costs for Actions

The **total cost** for an action consists of a summation of 2 parts:

- **movement cost** (same as part A)
- **floor cost** (value of the **destination** cell the robot is moving into)

This means although you may incur less movement cost to move straight to a target location, the additional floor cost along the way may be such that taking a roundabout way will result in an overall lower cost.

For example the lowest cost route to box 1 is not ['move e', 'move e']:

```
warehouse = ['*..1',
             '....',
             '.##.']

warehouse_cost = [[ 1, 100,  50, 1],
                  [ 1,   1,   1, 1],
                  [ 1, inf, inf, 1],]
```

Two example calculations for the **total cost** of an action using the example grid above are:

- If the robot enters (0,1) from (0,0) then the total action cost will be:
  total cost = ***horizontal*** **movement cost** + **destination floor cost** = ***2*** + 100 = 102.

- If the robot enters (0,1) from (1,0) then the total action cost will be:
  total cost = ***diagonal*** **movement cost + destination floor cost** = ***3*** + 100 = 103.

Note that the **floor cost** to move into cell (0,1) is 100 regardless of the direction the robot is entering from.

Three example calculations for the **total cost** of **illegal** actions (i.e. attempting to move to (or put down a box at) an occupied space or outside the warehouse) are:

- If the robot attempts to move east from (2,0) then the total action cost will be: **total cost** = ***horizontal*** **movement cost + illegal movement cost** = 102.
- If the robot attempts to move southeast from (2,0) then the total action cost will be: **total cost** = ***diagonal*** **movement cost + illegal movement cost** = 103.
- If the robot attempts to put down a box to the southeast from (2,0) then the total action cost will be: **total cost** = **put down box cost + illegal movement cost** = 102.

Note that the movement costs are still included in the case of illegal actions even though they weren't successful (the robot still exerted the energy). Floor costs are only incurred when the robot legitimately enters a square. Floor costs (of the box location) are incurred when lifting/putting down boxes.

Part B Method Return Specifications

`plan_delivery` should return two policies, each as a `list` of `lists` of `strings` indicating the action to take at each square on the grid. The format of the commands is the same as in part A. The special command '-1' should be placed at any square for which there is no valid command, such as a wall.

For example, for the values of `warehouse` and `todo` given previously (reproduced below):

```
warehouse = ['1..',
             '.#.',
             '..@']
```

`plan_delivery` might return the following two policies:

To Box Policy:

```
[['B'     , 'lift 1' , 'move w' ]
 ['lift 1',    '-1'  , 'move nw']
 ['move n', 'move nw', 'move n' ]]
```

Deliver Box Policy:

```
[['move e' , 'move se', 'move s']
 ['move ne',    '-1'  , 'down s']
 ['move e' , 'down e' , 'move n']]
```

where: 'B' indicates the box location.

For the "Deliver Box Policy", the dropzone includes an action in the event the robot starts on, lifts an adjacent box, and then must move off the dropzone to deliver it.

Part B Scoring

The testing suite will pick a starting location for the robot and then execute the actions specified by the "To Box Policy" until it finds and lifts the box. Then it will use the "Deliver Box Policy" and, given the location of the robot when it lifted the box, the appropriate commands are executed until the the box is delivered to the dropzone. The total cost of the student delivery is calculated: `student_cost`. The score for each test case will be calculated by: `benchmark_cost` / `student_cost`. The benchmark will be greater than or equal to the absolute minimum cost, which leaves an opportunity for extra credit on some test cases.

Part C (20%)

In this part there is only one main difference from part C:

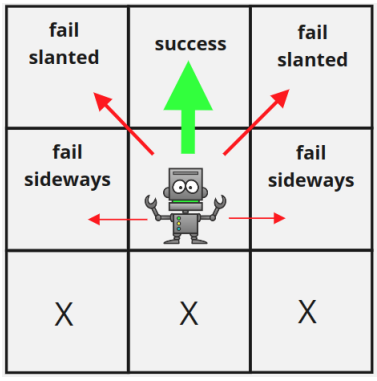
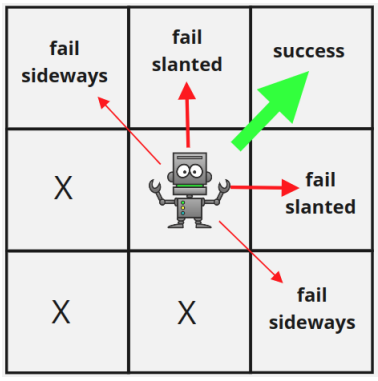
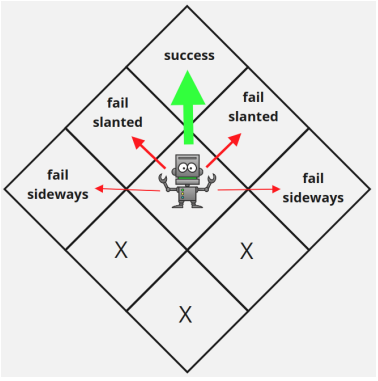
- `move` actions are stochastic

In part A and B we dealt with a deterministic robot. In real life however, we are inevitably faced with stochasticity. As such, part C is about finding an optimal policy based on stochastic robot movements.

Note: For this part you should find 2 individually optimal policies: pick up and drop off. This means your main algorithm should be executed 2 times: once to obtain the optimal policy to pick up the box and once to obtain the optimal policy to deliver the box.

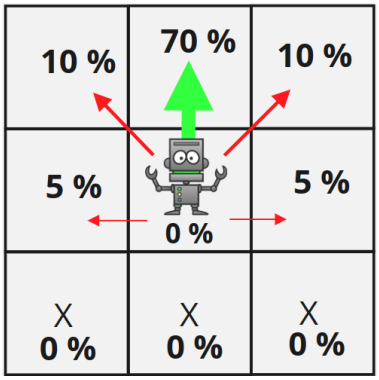
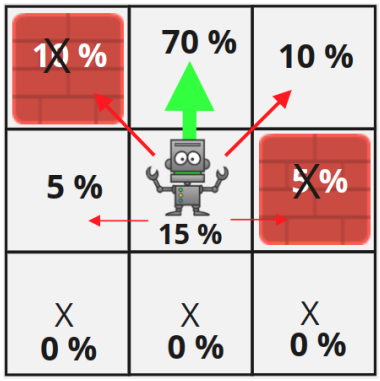
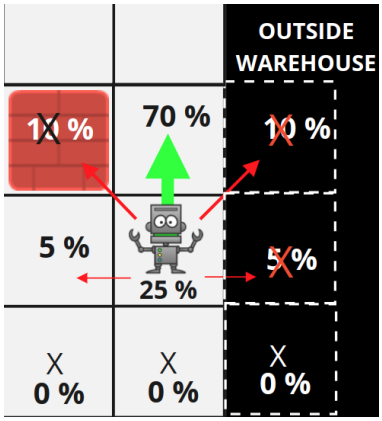
Part C Rules for Actions

Rules for actions are almost the same as part A & B. Instead of deterministic movements however, the robot will move according to a probability distribution defined by `p_outcomes`. `p_outcomes` will give you the probabilities of `success`, `fail_slanted`, and `fail_sideways` as depicted in the grids below. Your code should be able to handle any distribution provided to you in `p_outcomes`. `success` will be strictly greater than 0% and strictly less than 100%. The `success` direction in the images below indicates the intended action by the robot. Note that `fail_slanted` and `fail_sideways` are with respect to the intended directional movement.

Example #1	Example #2	Example #3
		

Note that Example 2 and 3 above are the same since orientation does not matter in this project, they are both provided to emphasize that the failure outcomes are with respect to the intended directional movement.

To understand the stochastic movement probability better, lets take a look at a few concrete examples. Assume the movement probability distribution is given as: success = 70%, fail slanted = 10%, and fail sideways = 5%. The probability distribution showing the outcomes of an intended movement of "move n" in 3 different scenarios are depicted below:

Example #4	Example #5	Example #6
		

Notice that in example #5, the two locations occupied by a wall prevent the robot from moving into those spaces and therefore the robot stays in place 15% (10% + 5%) of the time. Similarly, any attempt to move outside the warehouse will result in the robot staying in the same location (as seen in example #6).

Only directional movement is stochastic. The `lift` and `down` actions are deterministic.

Part C Costs for Actions.

Same as part B, with the clarification that the cost incurred is the cost of the action **actually performed**. This may differ from the action *attempted*, due to the stochastic nature of Part C.

For example, if the intended move is vertical, but the robot ends up failing slanted then the result will be a diagonal movement cost. Similarly, if the intended move is diagonal, but the robot ends up failing sideways then the result will be a diagonal movement cost.

Part C Output Specifications

Same as part B.

Part C Scoring

TL;DR: for each correct policy (to-box and to-dropzone) you will earn 0.5 points for a total of 1 point per test case. More details about the scoring are below, but not required to complete the project. There is no extra credit for this part.

The testing suite will initialize a robot at `robot_init`. The random number generator (RNG) will be seeded. Then actions will be simulated using your policy and the RNG. The actual action carried out may not be the same as the action specified in your policy due to stochasticity. The actual actions carried out are recorded as a list of actions: `student_actions`. You are given 0.5 points if `student_actions` match the `expected_actions`.

Note that before starting the to-zone policy procedure the testing suite will place the robot at location `robot_init2` and pick up the box. This is to allow you partial credit to earn points for a correct to-dropzone policy even if you failed the to-box policy.

Note that there is very little information that can be gleaned from analyzing `expected_actions`. This is not intended as a means of debugging for the students, rather as a way to grade a policy.

Environment Test

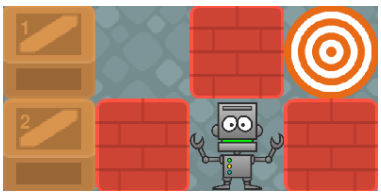
Before changing `warehouse.py`, test your environment using the following steps:

1) From the command line run: `python warehouse.py`
   - A list of moves for Part A test case 1 should be printed
   - A "to_box_policy" and "deliver_policy" will be printed for part B, test case 1
   - A "to_box_policy" and "to-zone_policy" will be printed for part C, test case 1
2) From the command line run: `python testing_suite_PartA.py` [or B, C]
   - A list of test cases and their score should show that test case 1 passed and the remaining failed.
   - There are more notes in `testing_suite_partA.py` to discuss how to run and debug

Visualization

There is an ASCII based visualization which will print the warehouse state and other important data to the console. This can be set by using the VERBOSE_FLAG in the testing suite files.

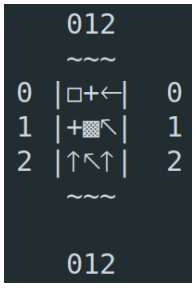
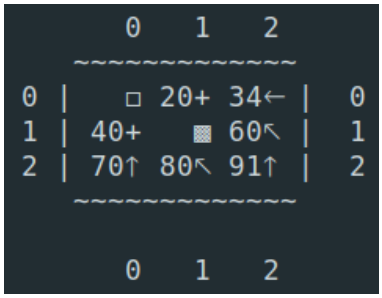
In addition, there is a GUI based visualization (only for part A), set VISUALIZE_FLAG=True. You can change the GUI frame rate speed in the **visualizer.py** file. The 6 choices are [1,2,3,4,5] (slow to fast) and [0] which is MANUAL-PAUSE mode (this will not proceed to the next time step until you press the **space bar**). You can conveniently quit any test case by pressing the `q` key.

GUI example screenshot	Illegal Move Indication
	

Development and Debugging

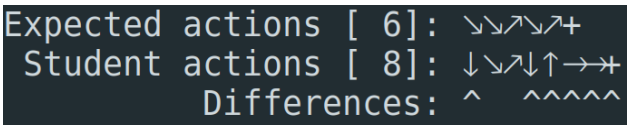
When developing and debugging here are some ideas that might prove helpful.

1) During initial development of your algorithm use warehouse.py and its main function
   - Copy a test case from the testing suite to the **main** in the bottom of warehouse.py
2) Test your algorithm using a single test case:
   - You can run a single test case. For example to run the first test case for partA:
     `python testing_suite_partA.py PartATestCase.test_case_01`
   - Or you may comment out all but a single test case in the testing_suite
3. If testing in a debugger, to allow breakpoints to work properly, there are some flags that can be set at the top of testing_suite_partA.py and testing_suite_partB.py
   - Set the TIME_OUT to a very large value (like 600 seconds)
   - Set DEBUGGING_SINGLE_PROCESS = True (this disables multiprocessing, which messes up most debuggers)
   - Set VERBOSE_FLAG = True
     - provides a simple console based visualization
     - provides line numbers for any syntax errors that occur
     - if exceptions are raised provides detailed stack trace
   - After the test case of interest works be sure to set the flags back to
     - VERBOSE_FLAG = False
     - TIME_OUT = 5
     - DEBUGGING_SINGLE_PROCESS = False
3) Part C outputs some additional terminal based data and visualizations that may be helpful in developing your solution, to turn them on set VERBOSE = True in the testing suite:

Symbol Policy	Values & Symbol Policy
	

Surrounding the warehouse policy are the row and column indexes so it is easier to locate a particular index (helpful on larger warehouses). The arrows denote the policy action. The empty square denotes a box. The white square denotes a wall. `+` denotes a `lift` command. `-` denotes a `down` command. Note that lift and down for part C are a little more lax as they do not check the box number nor direction.

If you also return a set of values to accompany your policies then these will be displayed (as integers) next to your actions. These values can represent anything you want and can serve as a way to visually see why certain actions are as they are.

Expected vs. Student Actions Comparison


The expected and student lists of actions are also output for part C. The difference between these are also marked with `^` indicating the place where the expected and student lists do not match.