CS 7638: Artificial Intelligence for Robotics

Warehouse Project

Summer 2021 - Deadline: Monday July 12th, Midnight AOE

Introduction

Your file must be called `warehouse.py` and must have three classes called:

1. `DeliveryPlanner_PartA`
2. `DeliveryPlanner_PartB`
3. `DeliveryPlanner_PartC`

- You may add additional classes and functions as needed provided they are all in this file `warehouse.py`.
- You may share code between partA and partB but it MUST BE IN THE FILE 'warehouse.py'
- Your submission will consist of the warehouse.py file (only) which will be uploaded to Gradescope. Do not archive (zip,tar,etc) it.
- Your code must be valid python version 3 code
- You may use the numpy library.
- Your `warehouse.py` file must not execute any code when imported.
- Ask any questions about these directions or specifications on Piazza.

Grading

- Your planner will be graded against a set of test cases, each equally weighted.
- If your planner returns a list of moves of total cost that is K times the minimum cost of successfully completing the task, you will receive 1/K of the credit for that test case.
- Otherwise, you will receive no credit for that test case. This could happen for one of several reasons including (but not necessarily limited to):
- `plan_delivery` output moves do not deliver the boxes in the correct order.
- `plan_delivery` creates output that is not a list of strings in the prescribed format.
- `plan_delivery` does not return an output within the prescribed time limit.
- Your code raises an exception.
- The weighting for each part is:
  - Part A = 40%
  - Part B = 45%
  - Part C = 15%

Part A (40%)

In this Part A, you will build a planner that helps a robot find the best path through a warehouse filled with boxes that it has to pick up and deliver to a dropzone.

`DeliveryPlanner_PartA` must have an `__init__` function that takes three arguments: `self`, `warehouse`, and `todo`. `DeliveryPlanner_PartA` must also have a function called `plan_delivery` that takes the argument, `self` and a flag `debug` set by default to `False`.

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 removed, the space functions as a `.` space.

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. Note that 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 contain empty space. (The dropzone is empty space)
- 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 for Movement

- Two spaces are considered adjacent if they share an edge or a corner.
- 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 robot may not move outside the warehouse.
- The warehouse does not "wrap" around (it is not cyclic).
- As described earlier, the robot may pick up a box that is in an adjacent square.
- The cost to pick up a box is 4, regardless of the direction the box is relative to the robot.
- While holding a box, the robot may not pick up another box.
- The robot may put a box down on an adjacent empty space (`.`) or the dropzone (`@`) at a cost of 2 (regardless of the direction in which the robot puts down the box).
- 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.

An illegal move will incur a cost of 100, and the robot will not move (the standard costs for a move will not be additionally incurred). 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 Output Specifications

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

- `'move {d}'`, where `'{d}'` is replaced by the direction the robot will 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 down the box: `"n", "e", "s", "w", "ne", "se", "nw", "sw"`

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 B (45%)

In this Part B, you will build a planner that helps a robot find the best path through a warehouse to a single box that it has to pick up and deliver to a dropzone. This part differs from part A, in that there is a single box, but now the warehouse has an "uneven" floor that imposes an additional, non-negative, cost on each robot command. In addition, the robot may "wake up" at any point in the warehouse and be tasked with retrieving the box and delivering it to the dropzone. Because of this, this project is most easily solved using the dynamic programming approach covered in Search:15.-19. and problem set 4, question 5.

`DeliveryPlanner_PartB` must have an `__init__` function that takes four arguments: `self`, `warehouse`, `warehouse_cost`, and `todo`. `DeliveryPlanner_PartB` must also have a function called `plan_delivery` that takes the argument, `self` and a flag `debug` set by default to `False`.

Part B 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.
`1` : the single box to be retrieved. 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 removed, the space functions acts as a `.` space.

For example:

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

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.
- There is a wall in the warehouse cells in row 1, column 1.
- The remaining six warehouse cells contain empty space. (The dropzone is also empty space)

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, math.inf, 2],
                  [ 2,       10, 2]]
```

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

The argument `todo` is a list of alphanumeric characters giving the order in which the boxes must be delivered to the dropzone. For part B this is limited to a single box as follows:
`todo = ['1']`
which indicates that box `1` must be retrieved and delivered to the dropzone.

Note: `todo` is kept consistent with part A for convenience and possible future expansion of the project requirements.

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 Movement

Rules for Movement are the same as those for part A. Please refer to part A documentation.

Part B Costs for an Action

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

- **movement cost** (same as those for part A)

- **floor cost** (value of the destination cell the robot is moving into)

This means that although you may incur less movement cost to move straight to a target location, the additional floor cost 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',
             '....']
todo = ['1']

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

Two example calculations for the **total cost** of an action using the example 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.

One example calculation for the **total cost** of an illegal action (ie. moving into a box or outside of the warehouse) using the same example above is:

- If the robot attempts to move north from (0,0) then the this will be an illegal move (outside the warehouse) so the total action cost will be: **total cost** = **illegal movement cost** = 100.

Note that the cost to move north (vertical movement cost) is **not** included because this movement is illegal.

Part B Output Specifications

`plan_delivery` should return two policies, each as a LIST of LISTs of strings indicating the command 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..',
             '.#.',
             '..@']

todo = ['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. '-1' indicates a square with no command

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

The testing suite will pick a starting location for the robot and then execute the appropriate moves in the "To Box Policy" until it finds and lifts the box with a `lift 1` command. At this point execution transitions to 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.

Part C (15%)

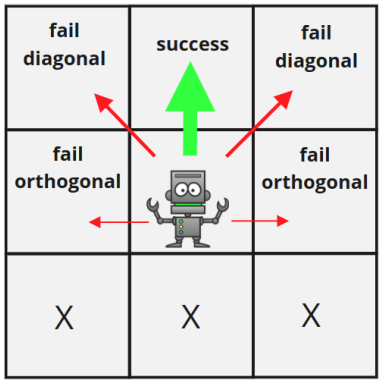
Part C is an extension to part B. In part A and B we dealt with a deterministic robot. In real life however, robot's don't always do what you tell them. Therefore, part C will be about finding an optimal policy based on stochastic robot movements.

Note: For this part you should find 2 individually optimal policies: pick up and drop off. This means your 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.

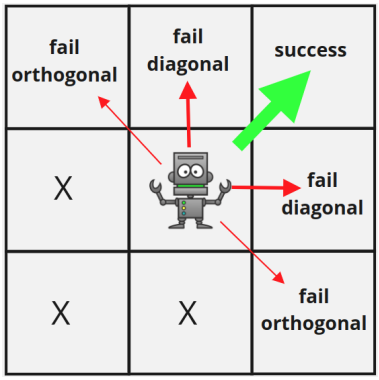
Part C Rules for Movement

Rules for movement 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_diagonal`, and `fail_orthogonal` 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.

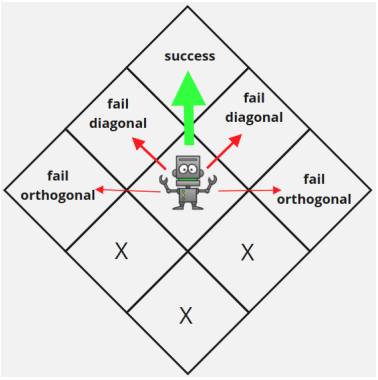
Example #1



Example #2

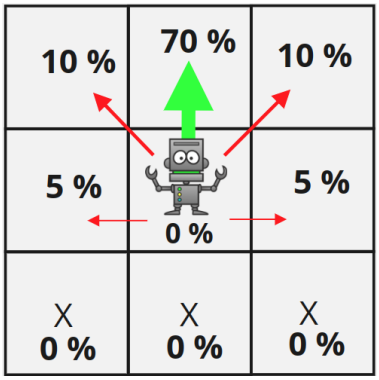
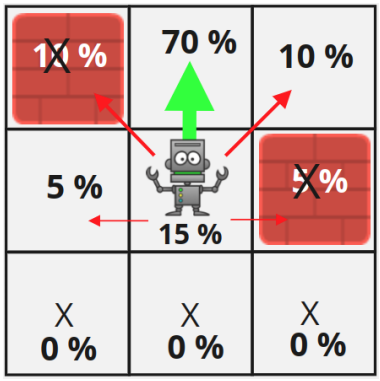
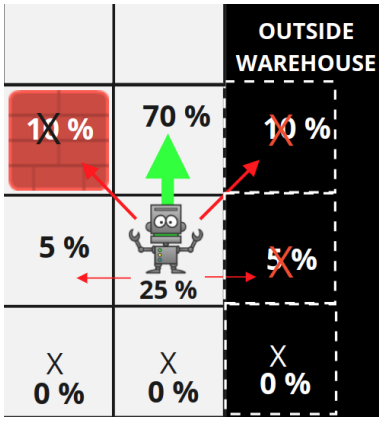


Example #3



Note that Example 2 and 3 above are the same since orientation does not matter in this project.

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 diagonal = 10%, and fail orthogonal = 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..

Note that only directional movement is stochastic. The `lift` and `down` actions are deterministic.

An illegal move will incur a cost of 100, and the robot will not move (the standard costs for a move will not be additionally incurred). Illegal moves include:

- attempting to move to a nonadjacent, nonexistent, or occupied space
- attempting to pick up a nonadjacent or nonexistent box
- 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 C Costs for an Action.

Same as part B, with the cost incurred being 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 orthogonal, but the robot ends up failing diagonally, the movement cost is that of the diagonal movement.

Part C Output Specifications

Same as part B.

Part C Scoring

The TL;DR version is if you have a correct to-box policy you will earn 0.5 points. If you have a correct to-dropzone policy, you will earn 0.5 points. This comes out to a total of 1.0 point for each test case. More details about the scoring are below, but not required to complete the project.

There are 10 test cases in part C. Each test case is worth 1 point. You can earn 0.5 points for each correct action series produced. Since there are only 2 polices (to-box and to-dropzone), each producing 1 action series, there is a total of 1.0 point available for each test case (no extra credit). So what is an action series? The testing suite will set a seed for a random number generator to obtain consistent results for each run of your policy. This seed value will not be known to you at test time. It will then take your `to-box` policy along with a robot starting location `robot_init` and perform up to 40 stochastic actions based upon guidance from your policy. The number of actions may be less if the robot reaches the destination. The actual actions

carried out are recorded as a list of actions (called your action series). This list is compared to a list of actions (action series) generated from an identical process using an optimal policy as a guide. If the lists match, then you will receive 0.5 points for each policy (to-box and to-dropzone) for a total of 1 point for each test case. Note that before starting the to-box policy procedure the testing suite will start the robot at location `robot_init2` and pick up the box. This is so that you are still able to earn points for a correct to dropzone policy even if you failed the to-box 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
2) From the command line run: `python testing_suite_PartA.py` 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 it [This step can be used to test part B or C as well by changing `A` above.]

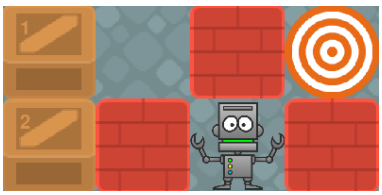
Visualization

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 (part A only VISUALIZE_FLAG=True). If you don't have this installed (say from using rait_env.yml) then you will need to install it before using the GUI: `pip install pygame`

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**)

GUI example screenshot



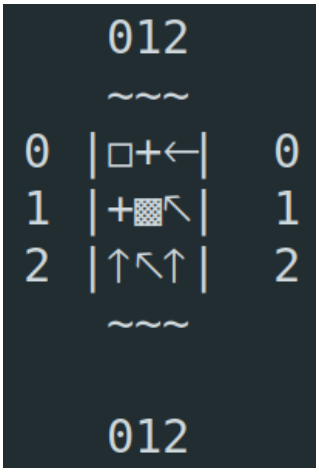
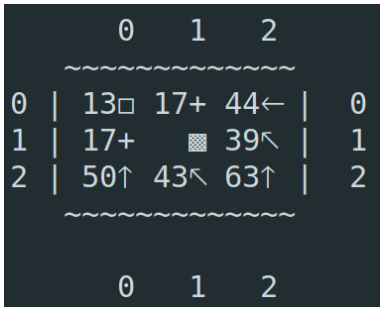
Development and Debugging

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
   - To run from the command line type: `python warehouse.py` or run in your IDE such as pycharm
   - Make sure code executes without errors
   - Check the output, moves (for part A) and policy (for part B)
2) Test your algorithm using testing_suite_partA.py (or B or C)
   - 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 flag TIME_OUT to a very large value (like 600 seconds)
- Set flag DEBUGGING_SINGLE_PROCESS = True (this disables multiprocess which messes with most debuggers
- Set the flag 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

4) 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.

Action Series Comparison

```
To zone:
Expected actions [ 5]: ↓↓→↡-
  Actual actions [ 9]: →↡↓↓→↡→→-
           Differences:  ^ ^  ^^^^^
```

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