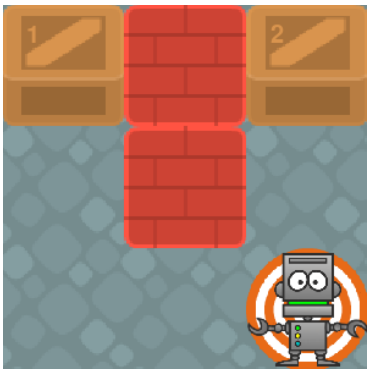
CS 7638: Artificial Intelligence for Robotics

Warehouse: Policy Search Project - Fall 2025 - Deadline: Monday Nov 17th, Midnight AOE



Introduction

This PDF serves as a problem specification document and thus the information in this document is not referring to any details regarding any specific algorithm to solve the problem. Its purpose is to define the problem along with the relevant details needed to solve it. It acts as a summary of the rules that are implemented in the testing suite code provided to you (which you can reference when you need further clarification). An interactive GUI mode (TEST_MODE) is also available for you to use (which you can use to validate your understanding of the specifications). Lastly, the example calculations and images in this PDF are also a good resource to use when looking for clarifications.

In this project, you will implement search algorithms to guide a robot through a warehouse to pick up and drop off boxes to a designated drop zone area.

The template code provides 2 classes, one for each part of the project: `DeliveryPlanner_Part[A, B]`

- You may share code between parts A and B
- Your submission will consist of a single file: `warehouse.py`

Grading

- The weighting for each part is:
  - Part A = 70%
  - Part B = 30%
- Within each part there are 10 test cases, each test case is equally weighted.
- No assumptions should be made about any similarities between local and GS test cases. All test cases will adhere to the rules laid out in this PDF. Your solution doesn't need to handle test cases that fall outside of these rules.

Part A (70%)

In this part, your task is to provide an optimal policy for a warehouse in which there will be only a single box for your robot to deliver. The warehouse has an "uneven" floor which imposes an additional cost (range: 0 ~ 95 inclusive). The robot starting location is not provided.

`DeliveryPlanner_PartA`'s constructor must have four arguments: `self`, `warehouse`, `warehouse_cost`, and `todo`.

Part A Input Specifications

`warehouse_viewer` will be a custom object used by the testing suite. For all intents and purposes, you can think of it as a list of `m` lists, each inner list containing `n` characters, corresponding to the layout of the warehouse. The warehouse is an `m` x `n` grid. `warehouse_viewer[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. There are NO restrictions on your code accessing the `warehouse` object. You may view as many cells as you wish.

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 (or exist at) 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 `.` (period).

`1` (box) : the single and only box in the warehouse to be delivered. 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, the space that the lifted box previously occupied now functions as a `.` (period).

Note: Test cases in the test suite will only contain the characters listed in Part A's input specifications section. 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 and may change it as you see fit, it is just provided for convenience. Note that the testing suite uses an asterisk (*) to denote the current location of the robot as it executes your plan. This asterisk is only used for internal purposes in the testing suite so you will not see it present in the test cases. It is also used to denote the robot's location in some examples in this document.

For example:

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

The argument `warehouse_cost` is a `list` of `lists` such that indices `i,j` refer to the floor 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, w, 2],
                  [ 2, 10, 2]]
```

where `w` represents a wall. Note that the value of `w` has no consequence since the robot can't occupy a space containing a wall.

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.

- Note: You must update your internal warehouse state in your code as this is not done for you.

Part A Rules and Costs for Motions

- The robot may move in 8 directions (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.
- There are 4 kinds of **motions** that the robot can take (below are the costs associated with each type):
  - [cost]: type
  - [2]: horizontal or vertical movement
  - [3]: diagonal movement
  - [4]: pick up box (regardless the direction)
  - [2]: put down box (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 end up in the same location when an illegal motion is performed.
- An illegal motion will incur a penalty cost of 100 in addition to the motion cost.
- Illegal motions 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 already holding one (attempting to put down a box while not holding one)
  - 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)

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

- **motion cost** (shown above)
- **floor cost**
  - *movements*: value of the **destination** cell the robot is moving into
  - *lift*: value of the cell the **box** is located in prior to lifting
  - *down*: value of the cell the **box** is being placed into

This means although you may incur less motion 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, 95, 50, 1],
                  [ 1,  1,  1, 1],
                  [ 1,  w,  w, 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 action cost = **motion cost (*horizontal* movement)** + **floor cost (*destination*)**
  = ***2*** + 95 = 97.
- If the robot enters (0,1) from (1,0) then the total action cost will be:
  total action cost = **motion cost (*diagonal* movement)** + **floor cost (*destination*)**
  = ***3*** + 95 = 98.

Note that the **floor cost** to move into cell (0,1) is 95 regardless of the direction the robot is entering from.

Three example calculations for the **total action cost** of **illegal** motions (i.e. attempting to move into (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 action cost = **motion cost (*horizontal* movement)** + **illegal motion penalty cost**

= **2** + 100 = 102.
- If the robot attempts to move southeast from (2,0) then the total action cost will be:
  total action cost = motion cost (*diagonal* movement) + illegal motion penalty cost
  = **3** + 100 = 103.
- If the robot attempts to put down a box to the southeast from (2,0) then the total action cost will be:
  total action cost = motion cost (*put down box*) + illegal motion penalty cost
  = **2** + 100 = 102.

Note that the motion costs are **still included** in the case of illegal motions even though they weren't successful (the robot still exerted the energy). Floor costs are **only** incurred when a motion is legally carried out. Floor costs (of the box location) are incurred when legally lifting, and floor costs (of the drop location) are incurred when putting down boxes.

Part A Method Return Specifications

`plan_delivery` should return two policies, each as a `list` of `lists` of `strings` indicating the motion to take at each square on the grid. 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

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 a motion in the event the robot starts on, lifts an adjacent box, a nd then must move off the dropzone to deliver it.

Part A Scoring

The testing suite will pick a starting location for the robot and then execute the motions 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 box is delivered to the dropzone. The total cost of the student delivery is denoted as: `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. You will receive a 0 in the following situations:

- your code views more warehouse cells than specified in the test case: `viewed_cell_count_threshold`
- your code takes longer than the prescribed time limit
- your method returns the wrong output format

Part B (30%)

In this part there is only one main difference from Part A:

- `move` motions are stochastic

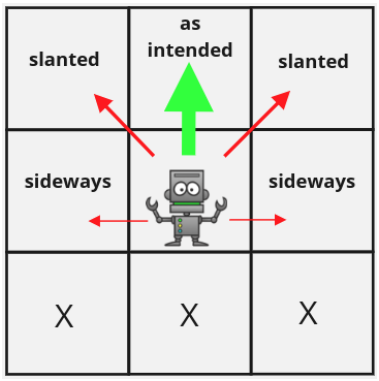
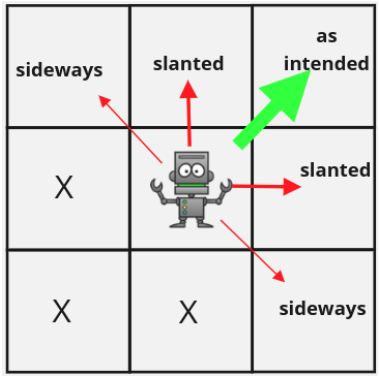
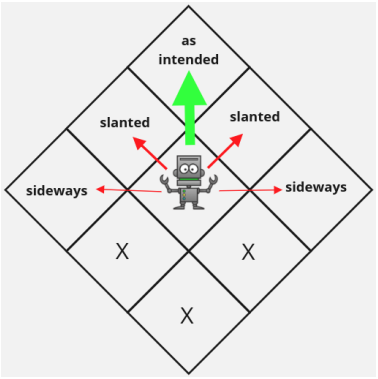
`DeliveryPlanner_PartB`'s constructor must have five arguments: `self`, `warehouse`, `warehouse_cost`, `todo`, and `stochastic_probabilities`.

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

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 B Rules for Motions

Rules for motions are almost the same as part A. Instead of deterministic movements however, the robot will move according to a probability distribution defined by `stochastic_probabilities`. `stochastic_probabilities` will give you the probability that the movement will be `as_intended`, `slanted`, or `sideways` as depicted in the grids below. Since these are probabilities, the sum of all possible outcomes will be one: $2*(sideways+slanted)+as\_intended=1$. Your code should be able to handle any distribution provided to you in `stochastic_probabilities`. `as_intended` will be strictly greater than 0% and strictly less than 100%. The `as_intended` direction in the images below indicates the intended movement by the robot. Note that `slanted` and `sideways` are **with respect to** the intended movement direction.

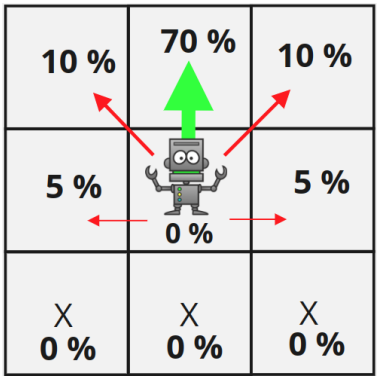
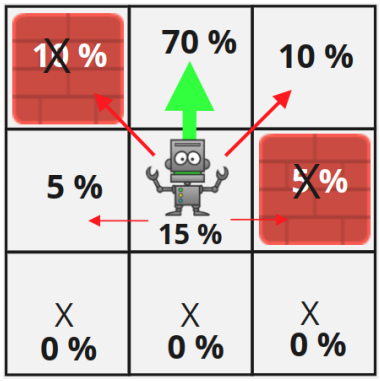
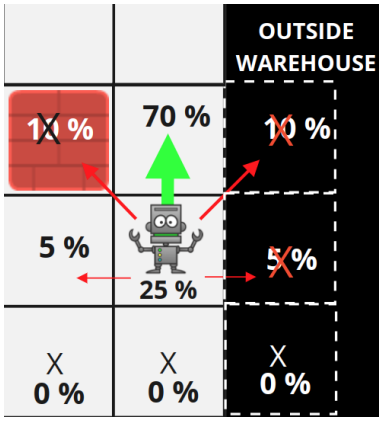
Example #1	Example #2	Example #3
		

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

To understand the stochastic movement probability better, lets take a look at a few concrete examples. Assume the movement probability distribution is given as:

- as intended = 70%
- slanted = 10%
- sideways = 5%

Do yourself a favor and validate that the sum of all possible outcomes for this example is indeed 1. 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` motions are deterministic.

Part B Costs for Actions

Same as part A, with the clarification that the cost incurred is the cost of the motion **actually performed**. This may differ from the motion *attempted* (intended), due to the stochastic nature of Part B.

For example, if the intended motion is a *vertical* movement, but the robot ends up performing a *slanted* movement then the result will incur a *diagonal* movement cost. Similarly, if the intended motion is a *diagonal* movement, but the robot ends up performing a *sideways* movement then the result will incur a *diagonal* movement cost. [This last statement is NOT a typo. Please reference the provided example images #1, #2, and #3 above to convince yourself.]

Part B Output Specifications

Same as part A.

Part B 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.

A random number generator will be seeded in order to produce deterministic (consistent) results. The testing suite will initialize a robot at `robot_init`. The student policy is then used to express the intended motion at each step. The performed motion (stochastic movement or deterministic lift/down) is recorded at each step. Note that the performed motion may not be the same as the motion specified in your policy (intended motion) due to stochasticity. The list of performed motions (actions) are recorded as: `student_performed_actions`. Since the random number generator is seeded, a particular policy will always produce the same list of performed motions. You are given 0.5 points if `student_performed_actions` match the `correct_performed_actions`.

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 `correct_performed_actions`. This is not intended as a means of debugging for the students, rather as a way to grade the policy. This

means you aren't given a crutch that tells you exactly what your code should be outputting, instead you must analyze your code by scrutinizing your implementation of the algorithm and making sure to adhere to the warehouse rules laid out in this document.

Environment Test

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

1) From the command line run: `python warehouse.py`
   - A "to_box_policy" and "deliver_policy" will be printed for part A, test case 1
   - A "to_box_policy" and "to_zone_policy" will be printed for part B, test case 1
2) From the command line run: `python testing_suite_PartA.py` [or B, or full]
   - A list of test cases and their score should show that test case 1 passed and the remaining failed.

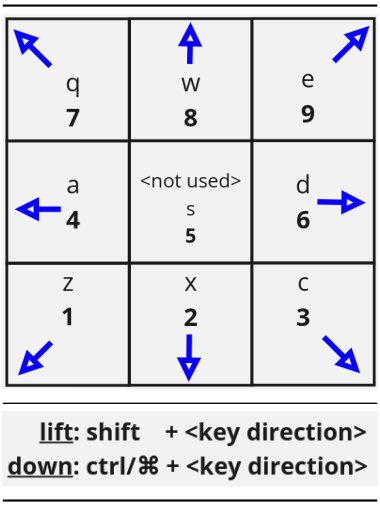
Visualization

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.

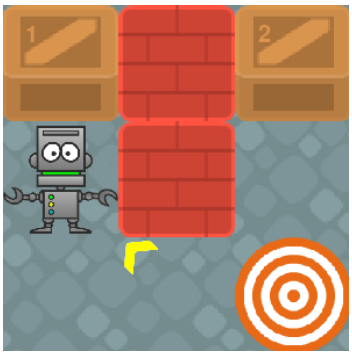
In addition, there is a GUI based visualization, set VISUALIZE_FLAG=True. You can change the GUI frame rate speed in the **visualizer.py** file. The 6 choices are [1,2,3,4,5] (slow to fast) and [0] which is MANUAL-PAUSE mode (this will not proceed to the next time step until you press the **space bar**). You can conveniently quit any test case by pressing the `Esc` (escape) key. An example demo video can be found at the following link: https://mediaspace.gatech.edu/media/t/1_onp8ge69

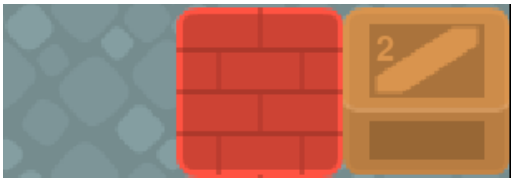
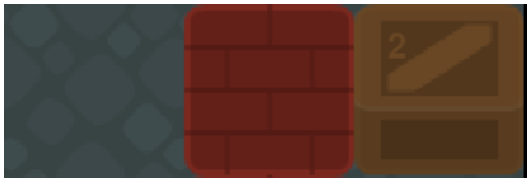
Part A's visualization will also indicate to you which cells your algorithm (did and didn't) access during the search process. The cells with a dark overlay on them indicate cells that your algorithm didn't access.

In `testing_suite_partA.py` you can turn on `TEST_MODE` to control the robot with your keyboard. This can be used to validate and build your understanding about the rules of the game. The controls are the following (note that NumLock may need to be turned off):



Illegal Move Indication	Illegal "box" Lift Indication	Illegal Move Example
		

Successfully Moved Diagonally	Successfully Put Box Down	Successfully Lifted Box
		

Cells Accessed	Cells Not Accessed
	

Development and Debugging

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

1. 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:

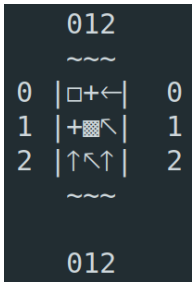
     ```
     python testing_suite_partA.py PartATestCase.test_case_01
     ```

   - Or you may comment out all but a single test case in the testing suite
2. If testing in a debugger, to allow breakpoints to work properly, there are some flags that can be set at the top of the testing_suite_part[A B].py files.
   - 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

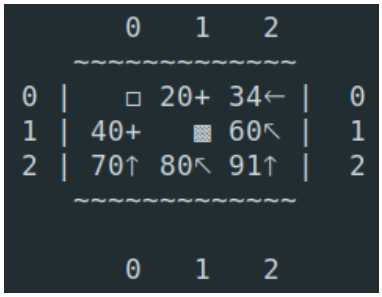
- provides a simple console based visualization
- provides line numbers for any syntax errors that occur
- if exceptions are raised provides detailed stack trace

3. Part B outputs some additional terminal based data and visualizations that may be helpful in developing your solution, to turn them on set VERBOSE_FLAG = True in the testing suite:

Symbol Policy



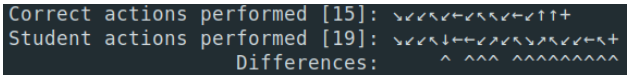
Values & Symbol Policy



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 motion. 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 motions. These values can represent anything you want and can serve as a way to visually see why certain motions are as they are.

Correct vs. Student Actions Comparison



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