A Bagful of Coins

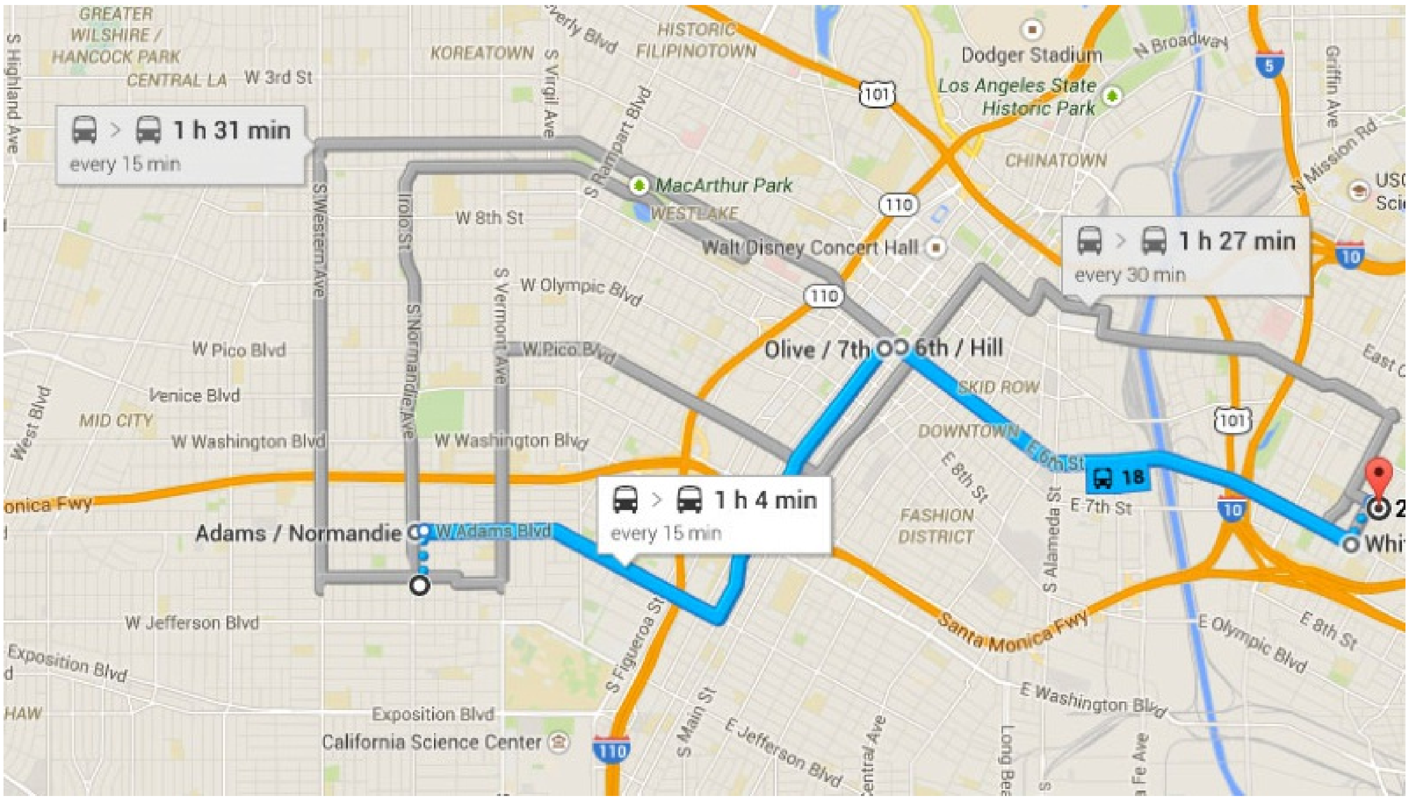
Submission to website: Monday, March 14, 10pm

Checkoff by LA/TA: Tuesday, March 15, 10pm

This lab assumes you have Python 2.7 installed on your machine. Please use the Chrome web browser.

Introduction

A GPS optimizes the path from a start to an end destination, using a variety of factors including traffic patterns, mileage, weather and more. It uses certain heuristics to make this search easier, like that freeways have higher speed limits than residential neighborhoods and that changing trains at the subway often takes added time.



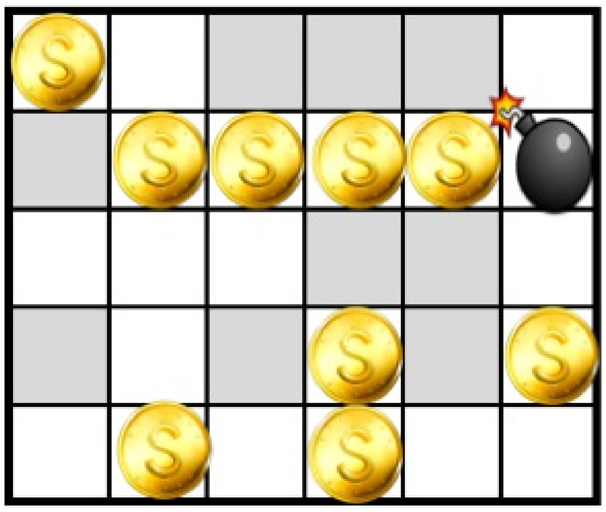
In this lab, you are going to implement a search algorithm analogous to to Google Maps. However, instead of optimizing for weather and traffic patterns in a city grid, you are going to be optimizing for coins in a fictional maze. Some neighborhoods in Boston are like mazes! Like for a GPS, it can be helpful to enumerate all of the valid paths from start to end destination, and this is what you are *required* to do.

You must find all of the paths through maze, collecting coins as you go. Be wary of bombs! They blow up all of your coins collected up to that point along the path. To make your life easier, you are only permitted to move down or right through the maze.

Maze Representation

In the problem, we represent mazes as `m` -by- `n` grids. Open space is represented with `0` , walls with `1` , coins with `“c”` , and bombs with `“b”` . You are free to move through open space, coins, or bombs, but cannot move through walls.

Below is an example, `maze_1` :



In the problem, mazes are represented as dictionaries. `maze_1` , from above, is represented as:

```
maze_1 : {
       "height": 5,
      "width": 6,
      "maze": [["c", 0,  1,  1,  1,  0 ],
               [ 1, "c","c","c","c","b"],
                 [ 0,  0,  0,  1,  1,  0 ],
               [ 1,  0,  1, "c", 1, "c"],
                [ 0, "c", 0, "c", 0,  0 ]]
   }
```

Notice that `maze_1[“maze”]` is an array of arrays, with a structure corresponding to the rows of the maze. To refer to specific coordinates, this lab assumes 0-indexing from the top left corner. Thus, the bomb in `maze_1` has coordinates `(5, 1)` .

Lab.py

To complete this lab, you must implement the `solve_maze` function. This function takes in a maze object (described above), along with coordinates of your start and goal positions.

Your goal is to return a maze object where each square corresponds to integer max amount of coins on valid path from `(start_x, start_y)` to `(goal_x, goal_y)`. If a square does not exist on a valid path from start to goal, then the corresponding square in the output maze should be an `"X"`. If you encounter a *bomb* along the path, then that resets the coin count to 0 for all paths going from the *bomb's* square to goal.

As mentioned above, you are only allowed to turn down and right through the maze. By default, each square with a wall in the input should return an `"X"` in the output, since you cannot move through walls. Even though a coin might be accessible moving down or right from the starting point, it only counts if it is located on a valid path to the the goal square. Remember to take into account that coins and *bombs* on on the start and goal squares are counted in the final coin tally.

In the event that no valid paths exist from the start to goal, your `solve_maze` implementation should return a maze of correct dimensions with all `"X"`'s.

The following is a concrete example:

Input:

- `maze_1`
- `(start_x, start_y) = (0,0)`
- `(goal_x, goal_y) = (5,4)`

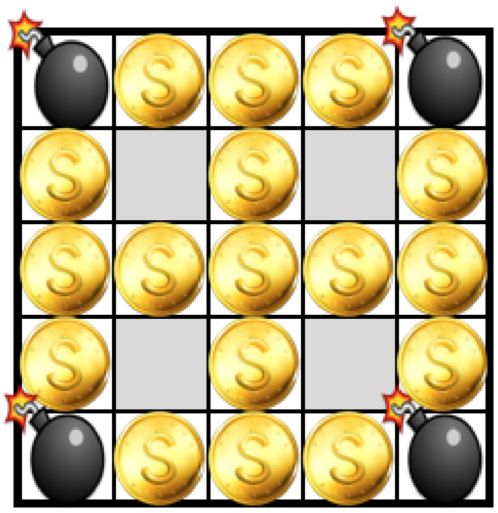
Output:

```
{
 “height”: 5,
 “width”: 6,
 “maze”:  [[ 1,  1, "X","X", "X","X"],
      ["X", 2,  3,  4,   5,  0 ],
      ["X", 2, "X","X", "X", 0 ],
      ["X", 2, "X","X", "X", 1 ],
      ["X", 3,  3,  4,   4,  4 ]]
}
```

1	1	X	X	X	X
X	2	3	4	5	0
X	2	X	X	X	0
X	2	X	X	X	1
X	3	3	4	4	4

In this case, there are exactly two valid paths from `(0,0)` to `(5,4)`, and integers representing coin counts along the paths are found in the output. Notice that in square `(5,1)`, the count for the path gets reset to `0`, since we encountered a *bomb*. Also notice that the value of coins in the goal square `(5,4)`, where the two paths merge, contains the maximum coin value along the two paths. Below is a slightly more complicated example:

Input:



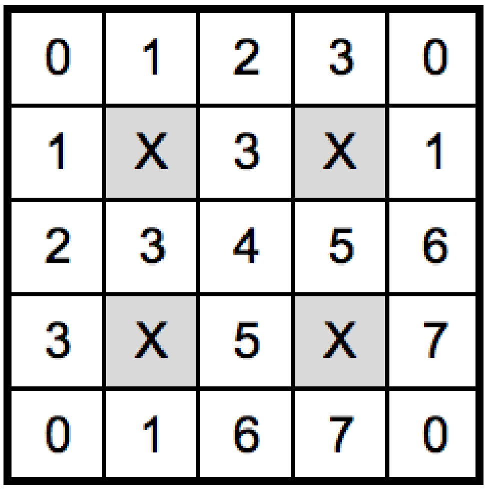
```
maze_2 = {
     "height": 5,
     "width": 5,
     "maze": [["b","c","c","c","b"],
              ["c", 1, "c", 1, "c"],
              ["c","c","c","c","c"],
              ["c", 1, "c", 1, "c"],
              ["b","c","c","c","b"]]
   }
```

- `maze_2`

- `(start_x, start_y) = (0,0)`
- `(goal_x, goal_y) = (4,4)`

Output

```
{
 'height': 5,
 'width': 5,
 'maze': [[0, 1, 2, 3, 0],
     [1,'X',3,'X',1],
    [2, 3, 4, 5, 6],
     [3,'X',5,'X',7],
     [0, 1, 6, 7, 0]]
}
```



0	1	2	3	0
1	X	3	X	1
2	3	4	5	6
3	X	5	X	7
0	1	6	7	0

We have a few interesting things happening in this example. First, we have bombs on both the start and goal squares. Ensure that your implementation correctly returns 0 on these squares. Next, we have certain squares with multiple valid paths passing through them. For instance, `(5,3)` has 3 valid paths passing through while `(4,4)` has 6. In the output, the value of these squares is the maximum value of coins collected along a path to the squares, or 0 in the case of a bomb. Finally, notice that the squares corresponding to walls in the input have `"X"` in the output, since no valid path exists through these squares to goal.

In-Browser UI (./server.py)

Once your `solve_maze` outputs a valid result, you can visualize the output to help you debug your code. Run `server.py` and open your browser to localhost:8000. You will be able to select the test case and visualize the maze and your paths.

Unit Tester (./test.py)

As in the previous labs, we provide you with a `test.py` script to help you **verify** the correctness of your code. The script will call the required methods in `Lab.py` and verify their output. In this lab, we encourage you to use the web UI to visually debug your simulation, and to **create your own** controlled test cases in order to verify your logic and diagnose your bugs. (You may need to clear your browser cache if you create your own test cases.) The `test.py` script will run all `.in` test cases in `cases/` .

We will only use the provided test cases to auto-grade your work. You may also wish to investigate the Python debugger (PDB) to help you find bugs efficiently.

Does your lab work? Do all tests in `test.py` pass? You're done! Submit your `Lab5.py` on funprog, and get your lab checked off by a friendly staff member. Consider tackling the bonus section below.

Bonus

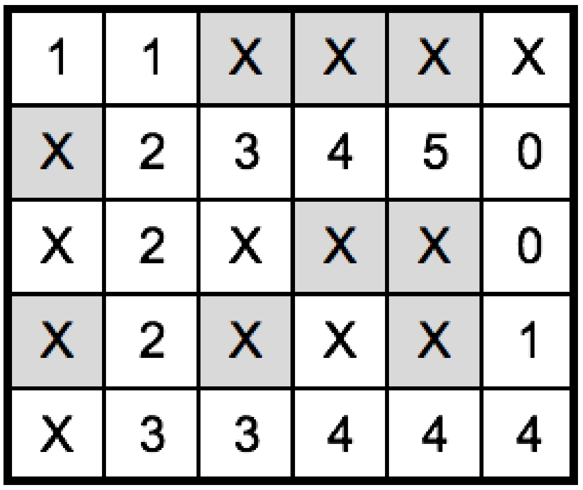
NOTE: This component is not a required portion of the lab.

Now that you have enumerated all of the paths, now we want to find the maximum weight path! This corresponds to the optimal navigation route through a city via GPS. Write a function `find_max_path` that takes in your output maze and gives the maximum weight path in tuples of coordinates from `(start_x, start_y)` to `(goal_x, goal_y)`. This function should output `None` if no path exists.

For example, back to `maze_1`:

Input:

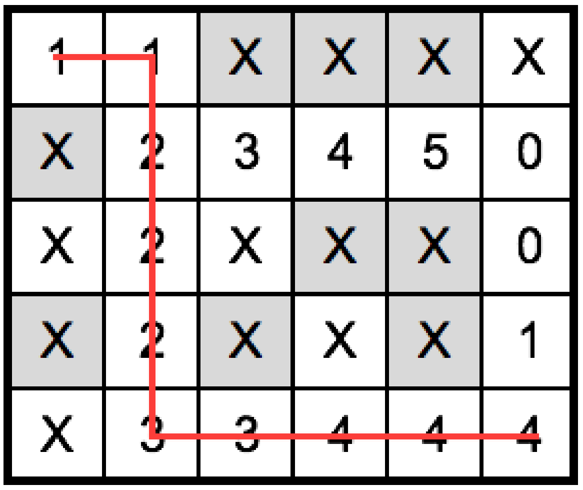
```
{
 “height”: 5,
 “width”: 6,
 “maze”:  [[ 1,  1, "X","X", "X","X"],
      ["X", 2,  3,  4,   5,  0 ],
      ["X", 2, "X","X", "X", 0 ],
      ["X", 2, "X","X", "X", 1 ],
      ["X", 3,  3,  4,   4,  4 ]]
}
```



1	1	X	X	X	X
X	2	3	4	5	0
X	2	X	X	X	0
X	2	X	X	X	1
X	3	3	4	4	4

Output:

```
[(0, 0), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4)]
```



1	1	X	X	X	X
X	2	3	4	5	0
X	2	X	X	X	0
X	2	X	X	X	1
X	3	3	4	4	4