Twists and Turns

Submission to website: Monday, April 18, 10pm

Checkoff by LA/TA: Tuesday, April 19, 10pm

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

Introduction

It's your birthday and your mom finally agreed to let you go to the arcade. Finally, you can win that giant fluffy unicorn you've been dreaming of your whole life! As you walk into the arcade, you see a massive claw machine with a joystick that can move your claw in multiple directions. This is it! Your chance to win that stuffed animal you've always wanted!

But wait! The joystick to control the machine looks pretty jammed: it looks like someone must have stuck their gum in there. Sometimes you can't get the joystick to turn left and other times you can't get the joystick to go straight! Thankfully, the arcade employees tell you about the joystick's quirky behavior after you put your quarter in the machine. They've been doing this a long time and can predict how many joystick moves it should take you to get your prize. If you go one move over the limit, the worker ends the game and you are left without a stuffed animal.

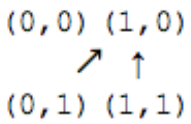
In this lab, we are going to implement an algorithm to solve all of our giant fluffy unicorn problems. Your assignment is to calculate the shortest path from the claw's initial position to the fluffy unicorn, in accordance with the worker's specifications. You may find the `graph.py` code from lecture very helpful (though you are free to ignore it).

Graph Representation

The space of all possible moves the claw can take is given as a graph encoded as a *list of directed edges* on a rectangular grid. Each *edge is a dictionary with keys "* `start` *" and "* `end` *"*, each of which denotes a location encoded as `[x,y]`. The top left corner of the grid is given by `[0,0]`, while the bottom right corner is `[m,n]` for a graph over an `m`-by-`n` grid.

A concrete example: the following edges represent the graph below.

```
edges = [{"start":[1,1], "end":[1,0] }, {"start":[0,1], "end":[1,0]}]
```

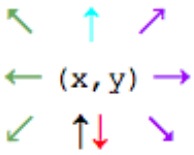


Grids may be very large and may contain edges between any two locations, including long, slanted edges. Each edge counts as exactly one move, even if it covers a lot of distance!

As in previous labs, the coordinate system in this lab places the origin `(0,0)` at the upper left corner, with the `X` axis extending to the right, and `Y` extending downward.

Defining Turns

Recall: in some cases, the claw is unable to go straight, while in others it is unable to turn to the left. When making a turn, you can't just look at where you're going; you also have to look at the direction you're coming from. In this lab, there are three relevant types of "turns".



To classify turns into "straight" and "left" ones, you may wish to calculate the cross-product and dot-product of two directions vectors (as given by the two graph edges that define a turn) - `d1` and `d2`. A turn from direction `d1` to direction `d2` is:

- "u-turn" if `cross_product(d1,d2)` equals 0 and `dot_product(d1,d2)` is less than 0 (red arrow). You are **never allowed** to do a U-turn.
- "straight" if `cross_product(d1,d2)` equals 0 and `dot_product(d1,d2)` is greater than 0 (cyan arrow).
- "left" if `cross_product(d1,d2)` is less than 0 (green arrows).
- "right" if `cross_product(d1,d2)` is greater than 0 (purple arrows).

Note: if you remember how cross products work, and the information above seems incorrect, recall that our coordinate system extends the `Y` axis downward. This changes the notion "left" and "right" from the standard cartesian corrdinate system.

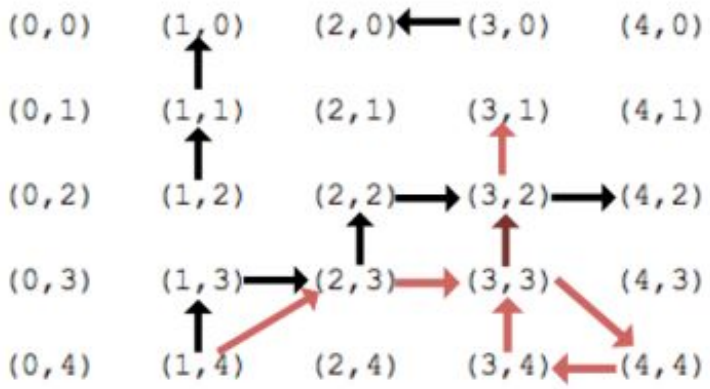
Searching for Paths

You have seen DFS (depth-first search without node marking) in lecture, and we invite you to modify `graph.py` from lecture to implement a recursive DFS to *enumerate all paths*, and select the shortest. If you look for a more efficient solution, consider implementing a breadth-first search (BFS, as you secretly used in the Bacon Number lab), which will enumerate paths in order of increasing length, finding the shortest path early.

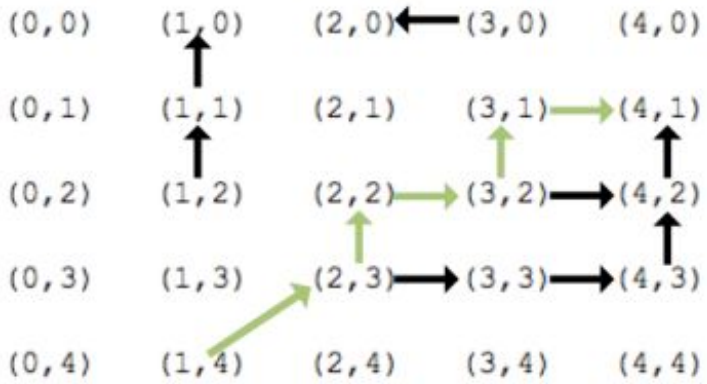
lab.py

You will implement your code in this file. You are not expected to read or write any other code provided as part of this lab. You must correctly implement the function `find_shortest_path(edges, source, destination, twisty)`, which is a function to find the shortest path between the claw and unicorn that meets the conditions specified. A path is defined by a list of unique edges that are a subset of edges in `edges`. A path may visit the same location multiple times, but may not contain duplicate edges!

- `edges` is a list of dictionaries, where each dictionary has keys `"start"` and `"end"` denoting `[x,y]` location as described previously
- `source` is a location `[x,y]` denoting the initial position of the claw
- `destination` is a location `[x,y]` denoting the location of the stuffed animal
- `twisty`: a Boolean (`True` / `False`) indicating whether straight or left turns are allowed. Precisely, `twisty` implies no straight turns are allowed, while `not twisty` implies the claw must never turn left. In other words, if `twisty` is `False`, all left turns are forbidden, while if `twisty` is true, straight turns are forbidden, but left and right turns are allowed.



The above graph shows a valid path in red if `twisty` is `False`. Note the dark red arrow: it cannot be taken the first time, as it would be a left turn. However, a combination of right turns (3 or more since they can be at different angles) can emulate a left turn.



The above graph shows a valid path in green if `twisty` is `True`.

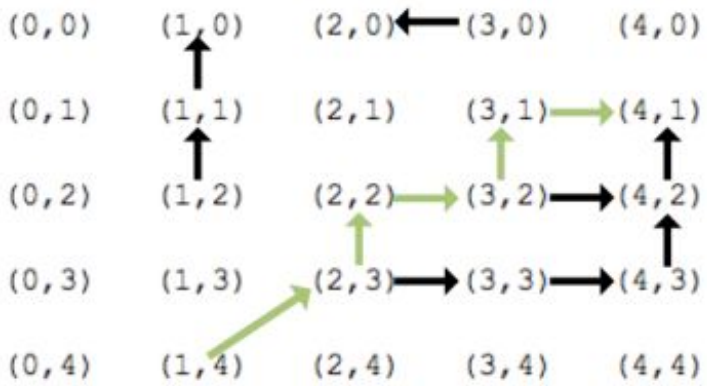
Your function should return the **shortest** path, if one exists, and `None` otherwise. The path should be a list of edges, each of which is a dictionary as defined above. Edges in a path must be edges in the graph, meaning they should be a subset of `edges`.

For example,

```
find_shortest_path([{"start":[1,1], "end":[1,0]},
                    {"start":[1,2], "end":[1,1]},
                    {"start":[3,0], "end":[2,0]},
                    {"start":[1,4], "end":[2,3]},
                    {"start":[2,3], "end":[3,3]},
                    {"start":[2,3], "end":[2,2]},
                    {"start":[2,2], "end":[3,2]},
                    {"start":[3,2], "end":[3,1]},
                    {"start":[3,2], "end":[4,2]},
                    {"start":[3,3], "end":[4,3]},
                    {"start":[3,1], "end":[4,1]},
                    {"start":[4,2], "end":[4,1]},
                    {"start":[4,3], "end":[4,2]}
                    ],
                   [1,4], [4,1], True)
```

would return

```
[{"start":[1,4],"end":[2,3]}, {"start":[2,3],"end":[2,2]},
    {"start":[2,2],"end":[3,2]}, {"start":[3,2], "end":[3,1]},
    {"start":[3,1],"end":[4,1]}]
```



Debugging your lab (./server.py)

Once your `find_shortest_path` 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 graph and your path. The source node (claw) is marked blue, while the destination node is red. As in the previous labs, we provide you with a `test.py` script to help you **verify** the correctness of your code, though it won't be of much help when debugging your implementation.

We will only use the provided test cases to auto-grade your work.

Does your lab work? Do all tests in `test.py` pass? You're done! Submit your `lab.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.

If you are done with the lab and would like another stuffed animal, we have another challenge for you. The arcade worker has been doing this for a while and knows a trick to get the claw to always be able to go straight or right AND make a **limited** number of left turns.

Implement the function `find_shortest_path_bonus(edges, source, destination, num_left_turns)` in the file `lab.py`.

The function `find_shortest_path_bonus` will be invoked with four parameters:

- `edges`: a list of dictionaries, where each dictionary has a 'start' list and an 'end' list as defined previously
- `source`: a list representing the original location of the claw
- `destination`: a list representing the location of the stuffed animal

- `num_left_turns` : an integer representing the maximum number of left turns allowed in the path

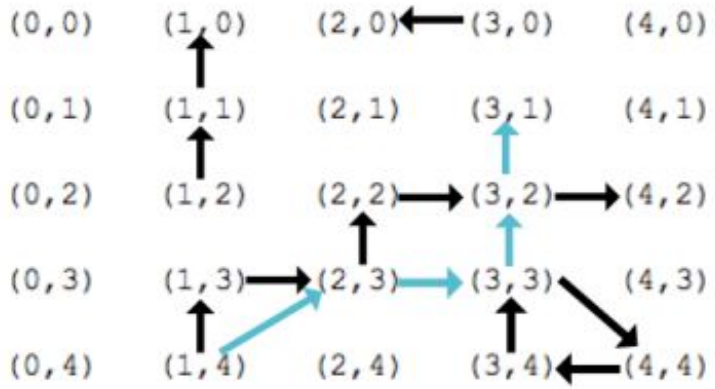
For example,

```
find_shortest_path_bonus([{"start":[1,1], "end":[1,0]},
                     {"start":[1,2], "end":[1,1]},
                     {"start":[3,0], "end":[2,0]},
                     {"start":[1,4], "end":[2,3]},
                     {"start":[1,4], "end":[1,3]},
                     {"start":[1,3], "end":[2,3]},
                     {"start":[2,3], "end":[3,3]},
                     {"start":[2,3], "end":[2,2]},
                     {"start":[2,2], "end":[3,2]},
                     {"start":[3,2], "end":[3,1]},
                     {"start":[3,2], "end":[4,2]},
                     {"start":[3,3], "end":[4,3]},
                     {"start":[3,3], "end":[3,2]},
                     {"start":[3,3], "end":[4,4]},
                     {"start":[4,4], "end":[3,4]},
                     {"start":[3,4], "end":[3,3]}
                     ],
                    [1,4], [3,1], 1)
```

would return

```
[{"start":[1,4],"end":[2,3]}, {"start":[2,3],"end":[3,3]},
    {"start":[3,3],"end":[3,2]}, {"start":[3,2], "end":[3,1]}]
```

as shown in the image below with the path in blue.



NOTE: We have provided test cases for the bonus section. They are in resources/cases_bonus. To use them, copy the files in resources/cases_bonus to the cases directory and test your code. These cases will work with the test script and with the UI.

HINTS

Please see the hints.pdf file if you want some help to get started.