# Lab 5: Don't Turn Left

## Table of Contents

## 1) Preparation

This lab assumes you have Python 3.7 or later installed on your machine.

The following file contains code and other resources as a starting point for this lab: `lab5.zip`

Most of your changes should be made to `lab.py`, which you will submit at the end of this lab. Importantly, you should not add any imports to the file.

This lab is worth a total of 4 points. Your score for the lab is based on:

- answering the questions on this page (1 points)
- passing the test cases from `test.py` under the time limit (2 points), and
- a brief "checkoff" conversation with a staff member to share your code, including discussion of the bullets below and a review of your code's clarity/style (1 points).

Please also review the collaboration policy before continuing.

**The questions on this page (including your code submission) are due at 4pm on Friday, October 18th.**

## 2) 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! 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 for 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.

## 3) Graphs

We can view the possible moves between configurations of the claw as a graph! As you probably already know, a *graph* is a set of *nodes* and *edges*. There are two kinds of graphs: graphs with *directed* edges, and graphs with *undirected* edges. Directed edges have a specified direction that points from one node to another node, while undirected edges simply connect two nodes without a direction.

In this lab, we will use nodes to represent *configurations* of the claw (where it currently is) and directed edges to represent possible moves of the claw (from one configuration to another) — edges are directed because some moves will be irreversible. All the nodes will correspond to lattice points on a rectangular grid. 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, with the $x$ axis extending to the right, and $y$ extending downward. Nodes are therefore represented as a tuple `(x,y)`, representing a lattice point with integer coordinates $(x,y)$ on the grid. An edge in this lab is represented as a dictionary with two keys `"start"` and `"end"` and two corresponding values which are nodes. A graph with multiple edges is represented as a list of edge dictionaries.

As a concrete example, the following `edges` list represents the graph drawn below.

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

$$\begin{tikzcd}
(0,0) & (1,0) \\
(0,1) \arrow[ur] & (1,1) \arrow[u]
\end{tikzcd}$$

Grids may be very large and may contain edges between **any two different locations**, including long, slanted edges. Each edge counts as exactly one move, even if it covers a lot of distance! For instance, an edge from `(0,0)` to `(1,0)` and another edge from `(0,0)` to `(7,0)`, while covering different distances, both count only as one move, if the claw moves along that edge.

It's important to think about what this graph really means. Each node represents the state of the claw (the coordinates of the claw's location), and each edge represents a move/transition from one state to another (moving from one coordinate to another). At any point in time, the claw will be on a grid square (state) and can use one of the outgoing edges (pointing away from the node) to move to a new grid square (transition). For some parts of this lab, this graph format will be exactly what we want. For later parts of the lab, we may want to make some modifications to the graph so that it better represents the different states and transitions that are present. Keep this in mind as you work on later sections of this lab.

### 3.1) Shortest Paths

Oftentimes, given a graph, we would like to know a shortest path (using the fewest edges) from one node (the "source") to another node (the "destination"), a problem that you may be familiar with from Lab 2! A common algorithm for this is **Breadth First Search (BFS)**. This algorithm grows shortest paths from the source node in order of increasing length, until finding the shortest path to the destination node. You may have used it to find Bacon paths in Lab 2 without even knowing! If this algorithm is new to you, take some time to try and understand how it works.

BFS explores different paths by gradually growing them outward. It first finds all nodes reachable by a length-1 path from the source node, then all nodes reachable by a length-2 path from the source node, and so on, until it finds a path to a destination node. Because BFS explores nodes starting with the nodes closest to the source node, it's guaranteed to find a shortest path. (It may also find alternate and possibly longer paths, so you need to take care to keep only the first path you find to a node.) You can visualize BFS as a ripple effect that occurs when a pebble is thrown into a pond; the pebble is the source node, and the ripples represent how BFS is exploring different paths in its efforts to try and find one to the destination. For more information about BFS, feel free to check out this guide or 6.006.

An important part of what makes BFS fast is **never considering an edge more than once**. Achieving this property requires some effort, as there may be *exponentially many* length-$k$ paths from the source node to a particular node. When you advance from all the nodes reachable by length-$(k-1)$ paths to compute the nodes reachable by length-$k$ paths, you need to remove any duplicates and any nodes from prior levels (e.g., using a `set`), or mark nodes when you first find a path to them and then ignore when you find other paths to marked nodes. If you don't do this, your algorithm could take exponential or even infinite time!

## 4) Debugging

Throughout the lab, when your functions are producing results, you can visualize the inputs and outputs to help you debug your code. Run `server.py` and open your browser to localhost:8000. You will be able to select a test case and visualize the graph and your path. The source node (initial location of the claw) is marked blue, while the destination node is red. The UI, unfortunately, does not display the direction of the edges. Some of the larger tests are too big to load into the UI, so you may notice their omission from the dropdown.

As in the previous labs, we provide you with a `test.py` script to help you verify the correctness of your code, though it may not be much help when debugging your implementation. We highly recommended **testing small examples** and **using the provided UI**.

## 5) Finding our Shortest Path

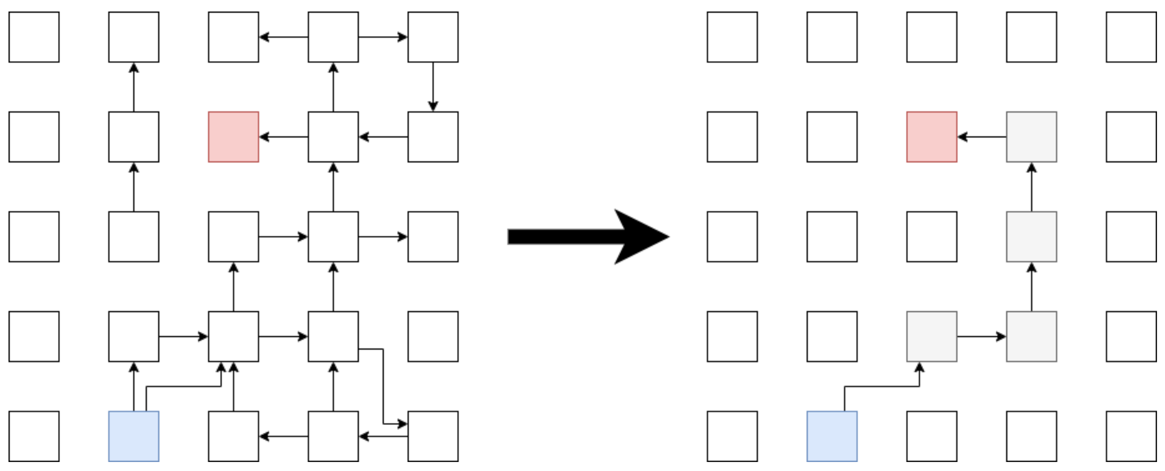
Now we're ready to try and win that fluffy unicorn! Sometimes, when we start up the machine, the employees tell us that everything is working correctly and we don't have to worry about the claw getting stuck. In these cases, all that's important is that we find the quickest way to get the claw from its starting position to the fluffy unicorn we want.

To help us figure out the best path in this situation, implement the function `shortest_path(edges, start, end)` in the file `lab.py`. The function `shortest_path` will be invoked with three parameters:

- `edges`: a list of dictionaries, where each dictionary has two items, with keys `"start"` and `"end"` and with values that are tuples (each with two integers), as defined above.
- `start`: a tuple representing the original location of the claw (drawn in blue).
- `end`: a tuple representing the location of the stuffed animal (drawn in red).

The function should return the list of edges taken along a shortest path if one exists. It should return `None` if it does not.

For example, given the input on the left, a valid solution is the path on the right.



This example is represented by the following call:

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

And the solution is represented by the following return value:

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

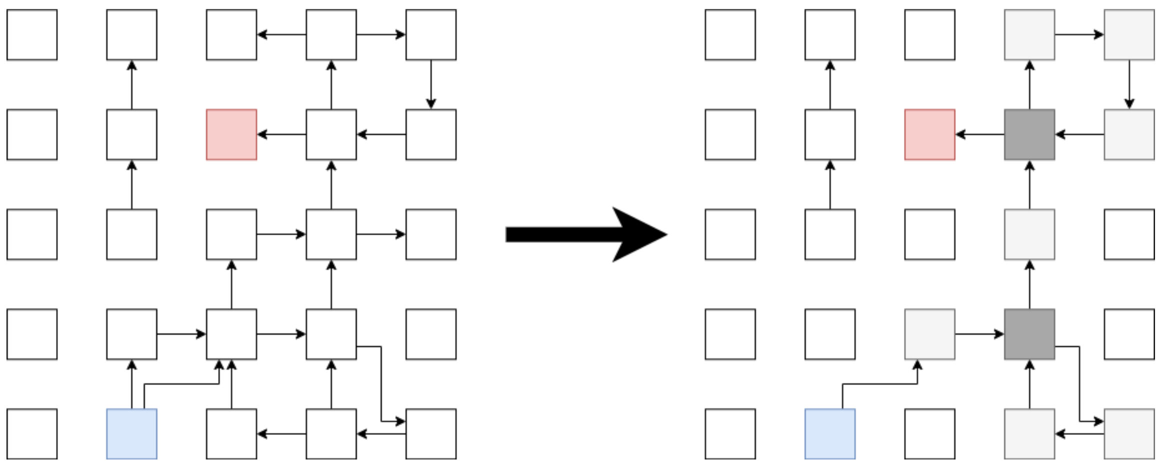
## 6) No Left Turns!

Unfortunately, fluffy unicorn acquisition won't always be so simple. Sometimes the gum in the machine is feeling extra sticky, and the claw won't listen to **any** of the left turns we tell it to do! The claw also refuses to go backwards, meaning we can't move forward from a square and then immediately go back to it. As a result, we need to find a shortest path that **doesn't have any left or reversing turns** in it.

To help us figure out the best path in this situation, implement the function `shortest_path_no_lefts(edges, start, end)` in the file `lab.py`. The function `shortest_path_no_lefts` will be invoked with same three parameters as `shortest_path`:

- `edges`: a list of dictionaries, where each dictionary has two items, with keys `"start"` and `"end"` and with values that are tuples (each with two integers), as defined above.
- `start`: a tuple representing the original location of the claw (drawn in blue).
- `end`: a tuple representing the location of the stuffed animal (drawn in red).

The function should return the list of edges taken along a shortest path if one exists. It should return `None` if it does not. For example, given the input on the left, a valid solution is the path on the right.



This example is represented by the following call:

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

And the solution is represented by the following return value:

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

**Check Yourself:**

The paths we find in this section may require us to visit some nodes more than once! We won't, however, move along the same edge twice. Why is that?

## 6.1) Turning Direction

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 four relevant types of "turns" to consider: left, right, straight, and U-turns.

To classify turns, you'll want to calculate the cross product and dot product of the two vectors covered by the edges. These can be computed by subtracting `start` from `end`, coordinate by coordinate, for both the previous edge and the currently considered edge. We call the resulting vectors `v1` and `v2` respectively.

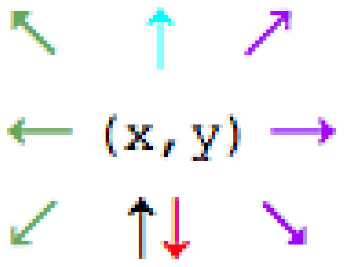
$$\text{cross\_product}(v_1, v_2) = v_1[0] \times v_2[1] - v_1[1] \times v_2[0]$$

$$\text{dot\_product}(v1, v2) = v_1[0] \times v_2[0] + v_1[1] \times v_2[1]$$

A turn from an edge with vector `v1` to an edge with vector `v2` can be classified as follows:

- If `cross_product(v1,v2)` equals 0, then
  - if `dot_product(v1,v2)` is less than 0, the direction is "u-turn" (red arrow). You are **not allowed** to do a U-turn;
  - otherwise, the direction is "straight" (cyan arrow).

- If `cross_product(v1,v2)` is less than zero, the direction is "left" (green arrows).
- If `cross_product(v1,v2)` is greater than zero, the direction is "right" (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 coordinate system.

**Note:** because we are not making a turn when we initially move, we can take any edge when we first start our path to the fluffy unicorn.

If you had started at `(0,0)` and had moved to `(5,3)`, what would be the resulting cross product when moving to `(3,4)`?

This question is due on Friday October 18, 2019 at 04:00:00 PM.

If you had started at `(0,0)` and had moved to `(5,3)`, what would be the resulting dot product when moving to `(3,4)`?

This question is due on Friday October 18, 2019 at 04:00:00 PM.

If you had started at `(0,0)` and had moved to `(5,3)`, what direction would you then be turning to go to `(3,4)`?

--

This question is due on Friday October 18, 2019 at 04:00:00 PM.

## 6.2) Transforming the Inputted Graph

Before, we could use a very generic BFS on the graph we were given to find a shortest path, which was really convenient. Now that isn't really the case. We need more information than what node we are on and what edges are available to truly know which ways we can move. Rather than finding a new way to get a shortest path, we can attack this problem by **transforming** the graph, that is, turning it into a graph that we can use regular BFS on! From here on out, we will call the graph that was given to us as input $G$ and the transformed "no-left-turn" graph $H_0$.

Recall from before that a graph is a way of representing states and transitions in that state. Previously, in $G$, the state of the claw was just the current location. We didn't care about anything that happened previously. Now the situation is more complicated. We need to also know where we came from to determine which of the edges we are allowed to take.

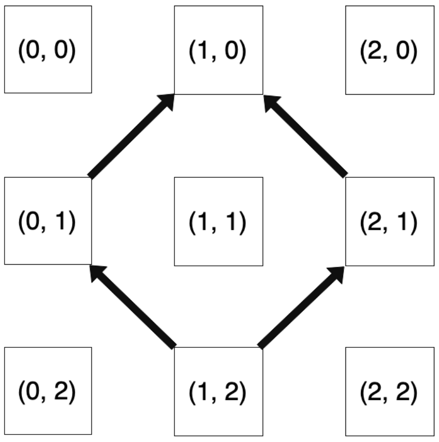
When we perform a transformation on this graph, the goal is to make one on which we can run BFS. This new graph will have a different set of states and transitions that represents the current "no left" pattern of movement better than the original graph.

Because we are making changes to the states and transitions, we can make sure we only include things that are possible.

As mentioned, we need to know both the current location and the previous location to determine which transitions we can take. This pair of current location and previous location may sound familiar because it is exactly an edge in our graph! This means that the set of nodes we want to use for $H_0$ are actually the edges in $G$.

For example, say that our graph $G$ includes the edges

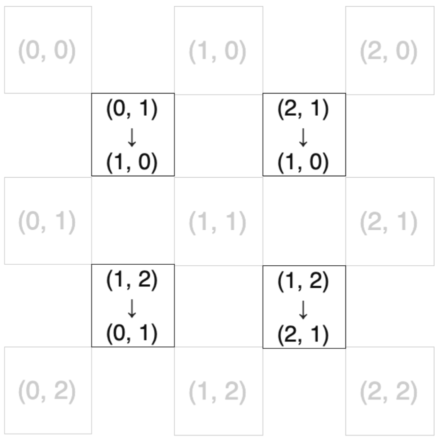
```
[
  {"start":(0,1), "end":(1,0)},
  {"start":(1,2), "end":(0,1)},
  {"start":(1,2), "end":(2,1)},
  {"start":(2,1), "end":(1,0)},
]
```



Graph $G$

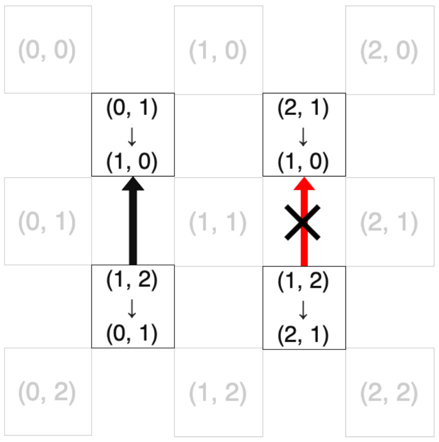
Then, if we were to similarly encode our states to be dictionaries with previous and current locations, the corresponding states in $H_0$ would look like

```
[
  {"previous":(0,1), "current":(1,0)},
  {"previous":(1,2), "current":(0,1)},
  {"previous":(1,2), "current":(2,1)},
  {"previous":(2,1), "current":(1,0)},
]
```



States of $H_0$

and the transitions of $H_0$ would consist of the legal, non-left turn moves between these states. In particular, for this set of states, we would have the transition `{"start":{"previous":(1,2), "current":(0,1)}, "end":{"previous":(0,1), "current":(1,0)}}` in the graph, but not the transition `{"start":{"previous":(1,2), "current":(2,1)}, "end":{"previous":(2,1), "current":(1,0)}}`, because this transition is a left turn.

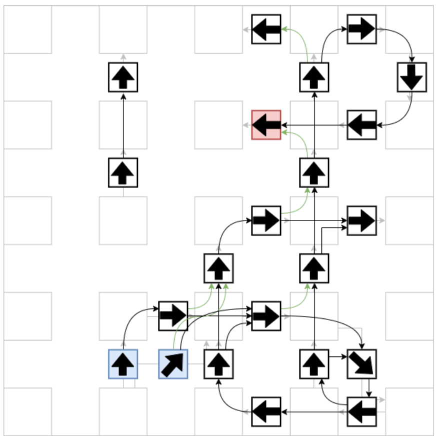


Graph $H_0$

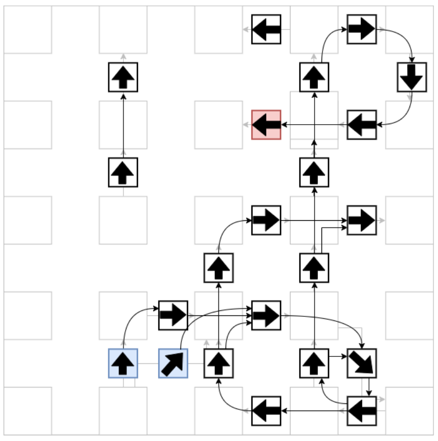
We can take a look at the what that transformation might look like on the example above.

We now have a node for each of the edges present in the graph. The arrows below demonstrate what direction the original edge moved in. Our new edges are between pairs of the original edges connected by a grid location. When we traverse the graph, we move from one original edge to another. This gives us enough information to determine what direction our turns actually are! The left turns are highlighted in green as before.

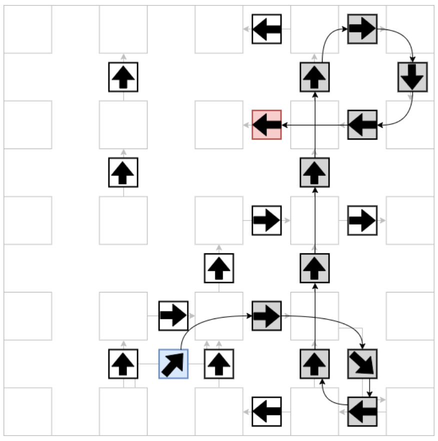
**Note**: Because there is more than one edge we can follow initially, we need to consider starting our BFS at both (and, in general, all) of those. If there were multiple edges that led into our target location `(2,1)`, we would need to consider multiple possible ending locations as well.



We can then remove the transitions that correspond to left turns to create the graph below. Every state and transition in this new graph is valid. Now, all we need to do is find a shortest path from start to end in this graph.



If we run BFS, one path we may get is the following. Because this path just consists of edges from the original graph, we can use it to reconstruct the path that was demonstrated earlier.



There are two main approaches to implementing this graph transformation:

- The first is to explicitly transform the graph we are given into a new graph. We fully construct each state from the list of edges, and each transition by finding which edges connect to one another. If we take care to not add transitions that are left turns, then we can simply run a regular BFS to find the desired result.
- The second approach is to implicitly traverse this graph by running a modified BFS on the original graph. This would consist of always keeping track of pairs of locations at each step. Each time we attempt a transition we can check whether or not it is valid by checking if it is a left turn. It may not seem like it's a graph transformation, but think about what it is you're actually adding onto your queue. When doing this implicitly, take care to make sure your code is not excessively repeated across sections!

## 7) Some Left Turns

Occasionally, when we start up the machine, the gum isn't fully stuck just yet. It may be pretty close though. This means we can make some left turns, but only a few before it does get stuck for good. Luckily the arcade employees at any point in time can tell exactly how many lefts are remaining. This means we need to find a shortest path that **has at most `k` left turns** in it. Like before, we are still **not allowed** to do U-turns.

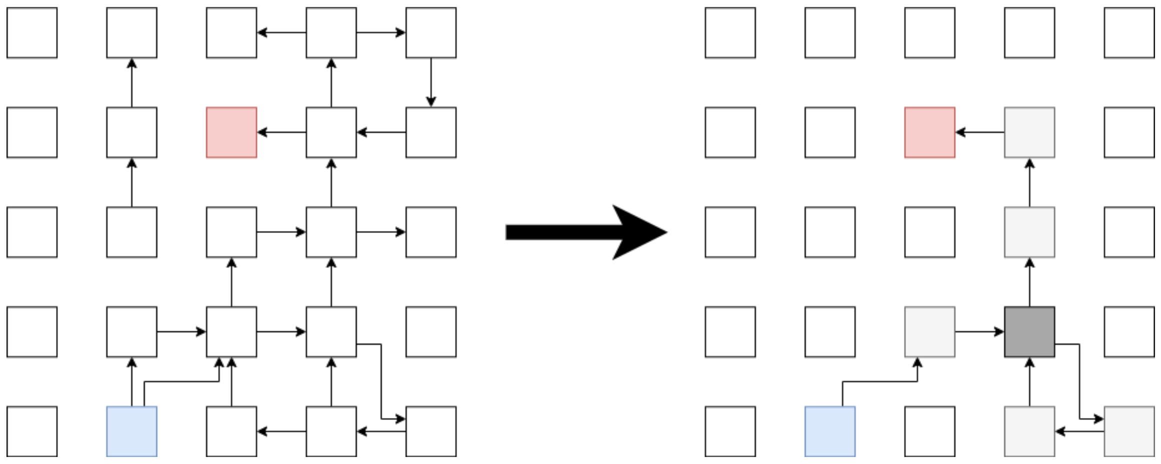
Implement the function `shortest_path_k_lefts(edges, source, destination, k)` in the file `lab.py`.

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

- `edges`: a list of dictionaries, where each dictionary has two items, with keys `"start"` and `"end"` and with values that are tuples (each with two integers), as defined above
- `start`: a tuple representing the original location of the claw
- `end`: a tuple representing the location of the stuffed animal
- `k`: an integer representing the maximum number of left turns allowed in the path

The function should return the list of edges taken along a shortest path if one exists. It should return `None` if it does not.

For example, given the input on the left, a valid solution is the path on the right.



This example is represented by the following call:

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

And the solution is represented by the following return value:

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

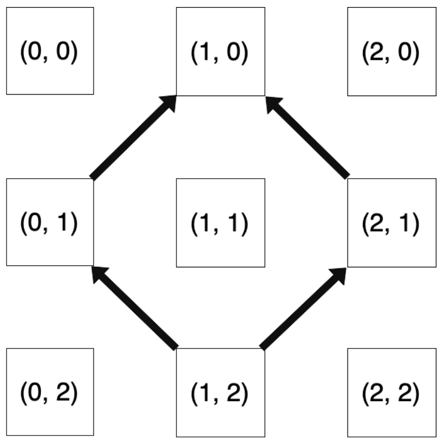
## 7.1) More Graph Transformations

Just like in our "no lefts" example, we can use a graph transformation to tackle this problem. Our state now not only consists of the current and previous locations, but also how many left turns we have taken so far! We will call the new graph from this transformation $H_k$.

In $H_k$, we let each state be composed of a `previous_location`, a `current_location`, and the number of `left_turns_remaining`. As in $H_0$, we should have nodes in $H_k$ that correspond to each edge of $G$, but this time we want to duplicate them for each possible value of `left_turns_remaining`, where `0 <= left_turns_remaining <= k`. We can then define a transition between two states as before, we just need to make sure we transition to the node in $H_k$ that has the correct number of `left_turns_remaining`. If the transition is not a left turn, we know that the number of `left_turns_remaining` is the same. If the transition is a left, we know that the number of `left_turns_remaining` will be decremented by one. Of course, if `left_turns_remaining = 0` then we can't take any of the transitions that would be left turns.

To look at an example, given the same simple graph $G$,

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



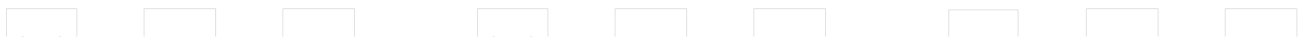
Graph $G$

and `k` having value 2, the states for $H_k$ could look like:

```
[
  {"previous":(0,1), "current":(1,0), "left_turns_remaining": 0},
  {"previous":(1,2), "current":(0,1), "left_turns_remaining": 0},
  {"previous":(1,2), "current":(2,1), "left_turns_remaining": 0},
  {"previous":(2,1), "current":(1,0), "left_turns_remaining": 0},

  {"previous":(0,1), "current":(1,0), "left_turns_remaining": 1},
  {"previous":(1,2), "current":(0,1), "left_turns_remaining": 1},
  {"previous":(1,2), "current":(2,1), "left_turns_remaining": 1},
  {"previous":(2,1), "current":(1,0), "left_turns_remaining": 1},

  {"previous":(0,1), "current":(1,0), "left_turns_remaining": 2},
  {"previous":(1,2), "current":(0,1), "left_turns_remaining": 2},
  {"previous":(1,2), "current":(2,1), "left_turns_remaining": 2},
  {"previous":(2,1), "current":(1,0), "left_turns_remaining": 2},
]
```



Some of the transitions in $H_k$ may be right turns, in which `left_turns_remaining` stays the same

```
[
  {
    "start":{"previous":(1,2), "current":(0,1), "left_turns_remaining": 0},
    "end":{"previous":(0,1), "current":(1,0), "left_turns_remaining": 0},
  },
  {
    "start":{"previous":(1,2), "current":(0,1), "left_turns_remaining": 1},
    "end":{"previous":(0,1), "current":(1,0), "left_turns_remaining": 1},
  },
  {
    "start":{"previous":(1,2), "current":(0,1), "left_turns_remaining": 2},
    "end":{"previous":(0,1), "current":(1,0), "left_turns_remaining": 2},
  },
]
```

and some would include left turns, in which `left_turns_remaining` goes down by 1:

```
[
  {
    "start":{"previous":(1,2), "current":(2,1), "left_turns_remaining": 1},
    "end":{"previous":(2,1), "current":(1,0), "left_turns_remaining": 0},
  },
  {
    "start":{"previous":(1,2), "current":(2,1), "left_turns_remaining": 2},
    "end":{"previous":(2,1), "current":(1,0), "left_turns_remaining": 1}
  },
]
```

Note what happened when we used the value `k = 2`; the graph from $H_0$ duplicated itself 3 times, one for each possible value of `left_turns_remaining`! This gives the new graph transformation for $H_k$ an intuitive three-dimensional representation, with `left_turns_remaining` forming the z-axis.

Which of the following best describes what the edge highlighted in orange above represents?

--

This question is due on Friday October 18, 2019 at 04:00:00 PM.

We can again look at the above example more closely to see how this graph transformation would lead us to an answer.

First we want to make `k + 1` copies of the graph, one for each possible number of left turns we can have remaining. For the example above, `k = 1` so we will have two copies. The copy on the right is for `left_turns_remaining = 0` and would be present in all $H_k$ where `k >= 0`. The copy on the left is for `left_turns_remaining = 0` and would be present in all $H_k$ where `k >= 1`.

We then use the left turns to connect them. Whenever we have a left turn, we transition between the copy for `left_turns_remaining = i` and the copy for `left_turns_remaining = i - 1`. This will lead to the graph below.

**Check Yourself:**

The graph with \`left_turns_remaining = 0\` is the exact graph we saw in the previous section! How could you use this fact to reduce repeated code?

**Check Yourself:**

The starting nodes always have \`left_turns_remaining = k\` but the ending nodes can be for any \`left_turns_remaining\`. Why is that?

Finally, as before, we can then run BFS on this graph to get the final solution. The states along this path are edges in the original graph along with extra information in the number of `left_turns_remaining`. We can again use the knowledge of what edges we followed to reconstruct what path we took in the original graph.

As before, this problem could be solved using an explicit or an implicit graph transformation:

- When explicitly transforming, we fully construct each state from the list of edges, `k + 1` times (each will include the current location, previous location, and number of left turns remaining). We can then make each transition by finding which edges connect to each other and have the correct number of left turns. Then, we can simply run a regular BFS to find the desired result.
- When done implicitly we run a modified BFS on the original graph that will always keep track of the current location, the previous location and the number of lefts turns remaining at each step. Each time we attempt a transition, we can check whether or not it is valid by checking if it is a left turn and that the number of `left_turns_remaining` is correct.

## 8) Code Submission

Once you have debugged your code locally and are satisfied, upload your `lab.py` below (Please note that if a function only returns `None`, that is considered hardcoding and you will not receive any points for that function even if it passes certain test cases):

Select File No file selected

**This question is due on Friday October 18, 2019 at 04:00:00 PM.**

## 9) Checkoff

Once you are finished with the code, please come to a lab session or office hour and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code in detail before asking for a checkoff.** Since the clarity of your code will be evaluated as part of the checkoff, you may wish to take some time to comment your code, use good variable names, create useful helper methods, etc. In particular, we will be checking for code repetition: you should not have three different implementations of BFS for this lab.

Be prepared to discuss:

- Your implementation of `shortest_path`
- Your implementation of `shortest_path_no_lefts`
- Your implementation of `shortest_path_k_lefts`
- What was similar and different about each of these three parts

## 9.1) Grade

*You have not yet received this checkoff. When you have completed this checkoff, you will see a grade here.*