CS 170 Homework 3

Due **Friday 9/20/2024, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, explicitly write "none".

2 Hadamard matrices

The Hadamard matrices $H_0, H_1, H_2, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix $[1]$
- For $k > 0, H_k$ is recursively defined as the $2^k \times 2^k$ matrix

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$$

For instance, the first three Hadamard matrices H_0, H_1, and H_2 are:

$$H_0 = \begin{bmatrix}1\end{bmatrix} \quad H_1 = \begin{bmatrix}1 & 1 \\ 1 & -1\end{bmatrix} \quad H_2 = \begin{bmatrix}1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1\end{bmatrix}$$

(a) Suppose that

$$\vec{v} = \begin{bmatrix}\vec{v_1} \\ \vec{v_2}\end{bmatrix}$$

is a column vector of length $n = 2^k$. $\vec{v_1}$ and $\vec{v_2}$ are the top and bottom half of the vector, respectively. Therefore, they are each vectors of length $\frac{n}{2} = 2^{k-1}$. Write the matrix-vector product $H_k\vec{v}$ in terms of H_{k-1}, $\vec{v_1}$, and $\vec{v_2}$ (note that H_{k-1} is a matrix of dimension $\frac{n}{2} \times \frac{n}{2}$, or $2^{k-1} \times 2^{k-1}$). Since H_k is a $n \times n$ matrix, and $\vec{v}$ is a vector of length n, the result will be a vector of length n.

Solution: $H_k\vec{v} = \begin{bmatrix}H_{k-1}\vec{v_1} + H_{k-1}\vec{v_2} \\ H_{k-1}\vec{v_1} - H_{k-1}\vec{v_2}\end{bmatrix} = \begin{bmatrix}H_{k-1}(\vec{v_1} + \vec{v_2}) \\ H_{k-1}(\vec{v_1} - \vec{v_2})\end{bmatrix}$

(b) Use your result from the previous part to come up with a divide-and-conquer algorithm to calculate the matrix-vector product $H_k\vec{v}$, and show that it can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations on scalars like addition and multiplication take unit time. You do not need to prove correctness.

Solution: Split the vector $\vec{v}$ in half into $\vec{v_1}$ and $\vec{v_2}$ as done in the previous part. Then compute $H_{k-1}(\vec{v_1} + \vec{v_2})$ and $H_{k-1}(\vec{v_1} - \vec{v_2})$ recursively, and concatenate the results to get $H_k\vec{v}$.

Alternate approach: after splitting the vector, one can instead compute $H_{k-1}\vec{v_1}$ and $H_{k-1}\vec{v_2}$ recursively and then add/subtract them as shown in the previous part to get the same result.

Runtime analysis: Let $T(n)$ represent the number of operations taken to find $H_k\vec{v}$. We need to find the vectors $\vec{v_1}+\vec{v_2}$ and $\vec{v_1}-\vec{v_2}$, which takes $O(n)$ operations. And we need to find the matrix-vector products $H_{k-1}(\vec{v_1}+\vec{v_2})$ and $H_{k-1}(\vec{v_1}-\vec{v_2})$, which take $T(\frac{n}{2})$ number of operations. So, the recurrence relation for the runtime is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Using the master theorem, this give us $T(n) = O(n \log n)$.

3 Distant Descendants

You are given a tree $T = (V, E)$ with a designated root node r and a positive integer K. For each vertex v, let $d[v]$ be the number of descendants of v that are a distance of at least K from v. Describe an $O(|V|)$ algorithm to output $d[v]$ for every v. **Please give a 3-part solution; for the proof of correctness, only a brief justification is needed.**

Hint 1: write an equation to compute $d[v]$ given the d-values of v's children (and potentially another value).

Hint 2: to implement what you derived in hint 1 for all vertices v, we recommend using a graph traversal algorithm from lecture and keeping track of a running list of ancestors.

Solution:
Observe that for a vertex v,

$$d[v] = \sum_{c \text{ is a child of } v} d[c] + (\# \text{ descendants of } v \text{ at distance } K)$$

The algorithm proceeds as follows. Initialize all the $d[v]$ values to 0 and start a dfs at the root node. At every step of the algorithm, we will maintain the ancestors of the current node in a separate array. To ensure that our array only contains vertices on our current path down the DFS tree, we'll only add a vertex to our array (at index equal to the current depth) when we've actually visited it. Since a path can have at most n vertices, the length of this array is at most n.

Now, while processing the node v, we first index into the array equal to the index of its kth ancestor, call this ancestor a. Then we increment $d[a]$ to account for v. Once we finish processing a child c of v, we increment $d[v]$ by $d[c]$.

We provide pseudocode for this algorithm below:

```
def find_distant_descendants(G=(V, E), r, K):
    d = [0 for v in V]
    visited = [False for v in V]
    ancestors = []

    def explore(v):
        visited[v] = True

        # if possible, increment the d-value of the Kth ancestor by 1
        if len(ancestors) >= K:
            d[ancestors[-K]] += 1

        # add v to the list of ancestors and recurse on children
        ancestors.append(v)
        for each edge(v, u) in E:
            if not visited[u]:
                explore(u)
        ancestors.poplast()
        # if possible, increment the d-value of the parent
        if len(ancestors) >= 1:
            d[ancestors[-1]] += d[v]

    explore(r)
    return d
```

Proof of Correctness: For a vertex v and a child c, every node counted in $d[c]$ should be included in $d[v]$ because their distance to v can only increase. Furthermore, nodes that are exactly K away from v will not be counted for any of its children, since they will be closer than K to the corresponding child. So, these get accounted for whenever we visit a kth descendant of v and increment $d[v]$. Notice that when we finish processing a node v, its $d[v]$ value will be correct and so it can be used by its parent.

Runtime Analysis: Since we perform a constant number of extra operations at each step of DFS, the algorithm is still $O(|V|)$.

4 Depth First Search

Depth first search is a useful and often efficient way to organize computations on a graph.

Let G be an undirected connected tree, and let $wt : E \to \mathbb{R}^+$ be positive weights on its edges. We show a template for depth-first search based computations below.

```
Input: Undirected connected tree $G = (V, E)$ and positive weights $wt(u, v)$ for each edge $(u, v) \in E$

Initialization:
$visited[v] \leftarrow False$ for all vertices $v$.
$L[v] \leftarrow 0$ and $M[v] \leftarrow 0$ for all vertices $v$.

function Explore(Vertex $u$)
    $visited[u] \leftarrow True$
    for each edge $(u, v)$ in $E$ do
        if NOT $visited[v]$ then
            PreVisit(u,v)
            Explore($v$)
            PostVisit(u,v)
```

DFS can be used for different purposes by defining the procedures PreVisit and PostVisit appropriately. In each of the following cases, write down pseudocode for PreVisit and PostVisit routines to perform the computation needed.

(a) For each vertex v, compute the maximum weight of an edge along the path from root r to vertex v and store it in array $L[v]$.

Solution:

```
procedure PreVisit($u, v$)
    $L[v] \leftarrow \max(L[u], wt[u, v])$

procedure PostVisit($u, v$)
    return
```

(b) For each vertex v, compute the maximum weight of any edge in the subtree rooted at vertex v and store it in array $L[v]$.

Solution:

```
procedure PreVisit($u, v$)
    return

procedure PostVisit($u, v$)
    $L[u] \leftarrow \max(L[u], L[v], wt[u, v])$
```

(c) For each vertex v, compute the depth of the tree rooted at vertex v, i.e., the length of the longest path from v to a leaf in its subtree, and store it in $L[v]$.

Solution:

```
procedure PreVisit($u, v$)
    return

procedure PostVisit($u, v$)
    $L[u] \leftarrow \max(L[u], L[v] + 1)$
```

(d) For each vertex v, compute the maximum degree among all the children of v and store it in $L[v]$

Solution:

```
procedure PreVisit($u, v$)
    $M[u] \leftarrow M[u] + 1$

procedure PostVisit($u, v$)
    $L[u] \leftarrow \max(L[u], M[v] + 1)$
```

5 Topological Sort Proofs

(a) A directed acyclic graph G is *semiconnected* if for any two vertices A and B, there is a path from A to B or a path from B to A. Show that G is semiconnected if and only if there is a directed path that visits all of the vertices of G. Make sure to prove both sides of the "if and only if" condition.

Hint: Is there a specific arrangement of the vertices that can help us solve this problem?

Solution: First, we show that the existence of a directed path p that visits all vertices implies that G is semiconnected. For any two vertices A and B, consider the subpath of p between A and B. If A appears before B in p, then this subpath will go from A to B. Otherwise, it will go from B to A. In either case, A and B are semiconnected for all pairs of vertices (A, B) in G.

Now we show that if G is semiconnected, then there is a directed path that visits all of the vertices. Consider a topological ordering $v_1, v_2, \ldots, v_n$ of the vertices in G. For any pair of consecutive vertices v_i, v_{i+1}, we know that there is a path from v_i to v_{i+1} or from v_{i+1} to v_i by semiconnectedness. But topological orderings do not have any edges from later vertices to earlier vertices. Therefore, there is a path from v_i to v_{i+1} in G. This path cannot visit any other vertices in G because the path cannot travel from later vertices to earlier vertices in the topological ordering. Therefore, the path from v_i to v_{i+1} must be a single edge from v_i to v_{i+1}. This edge exists for any consecutive pair of vertices in the topological ordering, so there is a path from v_1 to v_n that visits all vertices of G.

(b) Show that a DAG has a unique topological ordering if and only if it has a directed path that visits all of its vertices.

Remark: This means that a semiconnected DAG always has a unique topological ordering.

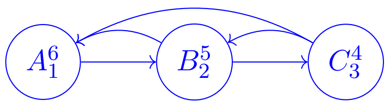
Solution: If a DAG has a directed path that visits all of its vertices, then arranging the vertices in the order they appear in the path will yield a topological ordering, as no backward edges can exist in this ordering since the graph has no cycles. There also clearly cannot be any other ordering as it would conflict with an edge of the directed path.

To prove the other direction, we'll proceed by contraposition. If a DAG does not have a directed path that visits all of its vertices, then by the previous part there exist two vertices A and B with no path between them. Then A and B can be interchanged and the resulting ordering will still be a valid topological ordering. Therefore, there exist at least two topological orderings so the topological ordering is not unique.

(c) This subpart is unrelated to the notion of semiconnectedness. Consider what would happen if we ran the topological sorting algorithm from class on a directed graph that had cycles.

Prove or disprove the following: The algorithm would output an ordering with the least number of edges pointing backwards.

Solution: The statement is false. Consider the possible DFS traversal on this graph yielding the following pre- and post-numbers:



The SCC-finding algorithm would output the ordering specified in the graph, which has 3 edges pointing backwards. However, ordering the vertices in the reverse order would yield an ordering with only 2 edges pointing backwards.

6 [Coding] DFS & Edge Classification

For this week's homework, you'll implement implement DFS and use DFS to classify edges in a graph as forward/tree, backward, or cross edges. There are two ways that you can access the notebook and complete the problems:

1. **On Datahub**: click here and navigate to the `hw03` folder.
2. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   `https://github.com/Berkeley-CS170/cs170-fa24-coding`

   and navigate to the `hw03` folder. Refer to the `README.md` for local setup instructions.

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the Gradescope assignment titled "Homework 3 Coding Portion".
- *Getting Help:* Conceptual questions are always welcome on Edstem and office hours; *note that support for debugging help during OH will be limited.* If you need debugging help first try asking on the public Edstem threads. To ensure others can help you, make sure to:
  1. Describe the steps you've taken to debug the issue prior to posting on Ed.
  2. Describe the specific error you're running into.
  3. Include a few small but nontrivial test cases, alongside both the output you expected to receive and your function's actual output.

  If staff tells you to make a private Ed post, make sure to include *all of the above items* plus your full function implementation. If you don't provide them, we will ask you to provide them.
- *Academic Honesty Guideline:* We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.

hw03

October 25, 2024

0.1 Depth First Search and Strongly Connected Components

0.1.1 If you're using Datahub:

- Run the cell below **and restart the kernel if needed**

0.1.2 If you're running locally:

You'll need to perform some extra setup. #### First-time setup * Install Anaconda following the instructions here: https://www.anaconda.com/products/distribution * Create a conda environment: `conda create -n cs170 python=3.10` * Activate the environment: `conda activate cs170` * See for more details on creating conda environments https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html * Install pip: `conda install pip` * Install jupyter: `conda install jupyter`

Every time you want to work

- Make sure you've activated the conda environment: `conda activate cs170`
- Launch jupyter: `jupyter notebook` or `jupyter lab`
- Run the cell below **and restart the kernel if needed**

[9]:
```
# Install dependencies
!pip install -r requirements.txt --quiet
```

[46]:
```
import otter
assert (
    otter.__version__ >= "5.5.0"
), "Please reinstall the requirements and restart your kernel."
import networkx as nx
import typing
import numpy as np
import tqdm
import pickle
grader = otter.Notebook("hw03.ipynb")

rng_seed = 42
```

[47]:
```
# Load test cases
file_path = "generated_testcases.pkl"
```

```
# Load the variables from the pickle file
with open(file_path, "rb") as file:
    loaded_data = pickle.load(file)
file.close()
inputs, outputs = loaded_data
```

Representing graphs in code There are multiple ways to represent graphs in code. In class we covered adjacency matrices and adjacency lists. There is also the edge list representation, in which you store the edges in a single 1 dimensional list. In general for CS170 and in most cases, we choose to use the adjacency list representation since it allows us to efficiently search over a node's neighbors.

In many programming problems, verticies are typically labelled 0 through $n-1$ for convenience (recall that arrays and lists in most languages begin at index 0). This allows us to represent an adjacency list using a list of lists that store ints. Given an edge list, the following code will create an adjacency list for an **unweighted directed graph**.

[48]:
```
def generate_adj_list(n, edge_list):
    """
    args:
        n:int = number of nodes in the graph. The nodes are labelled with␣
 ↪integers 0 through n-1
        edge_list:List[Tuple(int,int)] = edge list where each tuple (u,v)␣
 ↪represents the directed
            edge (u,v) in the graph
    return:
        A List[List[int]] representing the adjacency list
    """
    adj_list = [[] for i in range(n)]
    for u, v in edge_list:
        adj_list[u].append(v)
    for nodes in adj_list:
        nodes.sort()
    return adj_list

def draw_graph(adj_list):
    """Utility method for visualizing graphs

    args:
        adj_list (List[List[int]]): adjacency list of the graph given by␣
 ↪generate_adj_list
    """
    G = nx.DiGraph()
    for u in range(len(adj_list)):
        for v in adj_list[u]:
            G.add_edge(u, v)
    nx.draw(G, with_labels=True)
```

0.2 Q1.1) Reconstructing the DFS Path

In class we showed how to use DFS to check if there exists a path between two nodes, topologically sort nodes, and find SCCs. In those algorithms, pre and post numbers were used.

Here you'll implement a variation of DFS to print out the path between two nodes. In many problems, we want to be able to find the actual path between two nodes, not just determine if it exists.

> **Task:** Compute a path from s to t using DFS and return the path as a list of nodes on that path.

For example, the path $s \to a \to b \to c \to t$ corresponds to the list `[s, a, b, c, t]`. If no path exists, return the empty list `[]`.

You do not need to implement calculating pre and post numbers for this exercise.

Hint: 1. If you want to start with the recursive DFS implementation from DPV, you can use mutable types or the `nonlocal` keyword to preserve state across recursive calls. 2. It may be helpful to maintain an extra data structure which tracks the previous node we visited each time we visit a new node.

[49]:
```python
def dfs_path(adj_list, s, t):
    """
    args:
        adj_list:List[List] = an adjacency list
        s:int = an int representing the starting node
        t:int = an int representing the destination node

    return:
        a list of nodes starting with s and ending with t representing an s to␣
↪t path if it exists.
        Returns an empty list otherwise
    """
    def explore(adj_list, curr):
        """
        implements the explore subroutine from DPV, which is used in DFS. feel␣
↪free to delete this
        function and use an alternative implementation if you prefer.

        args:
            adj_list:List[List] = an adjacency list
            curr:int = the node currently being traversed

        return:
            None
        """
        # BEGIN SOLUTION
        nonlocal visited, prev # share the same visited and prev arrays across␣
↪all calls to explore()
```

```
        visited[curr] = True
        for v in adj_list[curr]:
            if not visited[v]:
                prev[v] = curr
                explore(adj_list, v)
        # END SOLUTION

    # implement the dfs and path reconstruction here
    # BEGIN SOLUTION
    # initialize
    n = len(adj_list)
    visited = [False]*n # an array of booleans representing if a vertex has␣
↪been visited
    prev = [-1]*n       # an array of ints representing the previous node on a␣
↪path from start to the current node

    # unlike DPV algorithm, only need to start the dfs from s
    explore(adj_list, s)

    # if t was not visited, then there is no path from s to t
    if not visited[t]:
        return []

    # if path exists, backtrack through the prev array to find the s-t path
    path = []
    curr = t
    while curr != s:
        path.append(curr)
        curr = prev[curr]
    path.append(curr)
    path.reverse()
    return path
    # END SOLUTION
```

0.2.1 Debugging

You can create sample tests in the following cells to help debug your solution. We provide a few small tests as an example, but they might not be comprehensive.

To add a new graph to the test, append a new edge list to `edge_lists` as shown in the next cell. **Remember that these edges are directed, so do not add both directions of an edge to the edge list.**

[50]:
```
edge_lists = []
edge_lists.append([(0,1), (0,2), (1,2), (2,3), (3,4), (3,5), (4,5)])   # edge␣
↪list of first graph
```

```
edge_lists.append([(0,1), (0,2), (1,2), (3,4), (3,5), (4,5)])           # edge␣
 ↪list of second graph
# add any additional tests here
```

For each test case you also need to add a starting node s, a destination node t, and n the number of nodes in the graph, add them to the following lists.

[51]:
```
s_list = []
s_list.append(0)  # s for first graph
s_list.append(1)  # s for second graph
# add any additional tests here

t_list = []
t_list.append(3)  # t for first graph
t_list.append(4)  # t for second graph
# add any additional tests here

n_list = []
n_list.append(6)  # n = 6 for first graph
n_list.append(6)  # n = 6 for second graph
# add any additional tests here
```

The following is a simplified version of the autograder, you may want to add more print statements or other debugging statements to check your function.

[ ]:
```
import matplotlib.pyplot as plt
index = 1
for s, t, n, edge_list in zip(s_list, t_list, n_list, edge_lists):
    print("Testing graph:", index)
    index += 1

    adj_list_graph = generate_adj_list(n, edge_list) # function defined earlier

    path = dfs_path(adj_list_graph, s, t)

    nx_graph = nx.DiGraph(edge_list)

    # uncomment the following to plot each graph
    '''
    nx.draw(nx_graph, with_labels=True)
    plt.title(f"Graph with {n} vertices and start node {s} and destination {t}")
    plt.show()
    '''

    if not nx.has_path(nx_graph,s,t):
        assert len(path) == 0, f"your dfs_path found an s-t path when there␣
 ↪isn't one."
```

```
    else:
        # checks that the path returned is a real path in the graph and that it␣
↪starts and ends
        # at the right vertices
        assert nx.is_simple_path(nx_graph, path), f"your dfs_path did not␣
↪return a valid simple path"
        assert path[0] == s, f"your dfs_path returned a valid simple path, but␣
↪it does not start at node s"
        assert path[-1] == t, f"your dfs_path returned a valid simple path, but␣
↪it does not end at node t"

print("Success")
```

[ ]:
```
grader.check("q1.1")
```

0.3 1.2) Pre and Post Numbers

In order to topologically sort or find strongly connected components, we need to be able to calculate pre and post numbers for each node.

In this part, you will rework your implementation of DFS to allow it to generate pre and post order numbers for each node. It might be a good idea to copy/paste your solution from the previous part and modify it here.

> **Task:** Implement a function that computes DFS pre and post numbers for each node in the graph.

To pass the autograder, your smallest preorder number should be 1. Your largest postorder number should be $ 2 ×(number of vertices)$. Return two lists of tuples, a `pre` list should containing tuples `(node, pre-number)`, and a `post` list containing tuples `(node, post-number)`.

Both lists should be ordered according to the pre/post number in the tuple. **You should not use any sorting functions to accomplish this!**

> **Reflect:** Why might returning pre/post numbers in this way be helpful for finding strongly connected components?

Feel free to delete the starter code and implement your own solution.

For this part, you can no longer assume that the entire graph is guaranteed to be reachable from some certain start node. How will this change your implementation?

Finally, break ties by choosing the node with the smallest number value. The autograder may fail implementations which are otherwise correct but break ties in a different way.

[54]:
```
def get_pre_post(adj_list):
    """
    args:
        adj_list:List[List[int]] = the adjacency list that represents our input␣
↪graph
    return:
        List[Tuple(int, int)], List[Tuple(int, int)] representing the pre and␣
↪post order values
            respectively. Each tuple should have a vertex as its first entry,␣
↪and the pre/post order
            value as its second entry.
    """
    time = 1
    pre = []
    post = []

    # YOUR CODE HERE
    # BEGIN SOLUTION
    n = len(adj_list)
    visited = [False]*n

    def explore(u):
        nonlocal time
        nonlocal visited
        visited[u] = True
        pre.append((u, time))
        time += 1
        for v in adj_list[u]:
            if not visited[v]:
                explore(v)
        post.append((u, time))
        time += 1
    for i in range(n):
        if not visited[i]:
            explore(i)
    # END SOLUTION

    return pre, post
```

[ ]:
```
grader.check("q1.2")
```

0.4 Q1.3 Identifying Tree, Forward, Back, Cross Edges

As we perform DFS traversals and create DFS trees and DFS forests within our graph, we would like to classify our edges according to how they appear in the resulting DFS forest. These classifications can provide us with insights about our graph. For example, the presence of a back edge (u, v) tells us that we have a cycle within this graph that includes all the tree edges on the path from v to u and the back edge (u, v).

> **Task 1.3** * Given the adjacency list of a graph, add each edge present in the edge set to the correct classification according the DFS traversal you implemented in part 1. * Don't modify the initialization of the edges_lookup dictionary.

```
[63]: def categorize_edges(adj_list):
          """
          args:
              adj_list:List[List[int]] = the adjacency list that represents our input␣
       ↪graph
          return:
              Dictionary({
                  'tree': set(),
                  'forward': set(),
                  'cross': set(),
                  'back': set()
              }) where each set() contains the edges that belong to the corresponding␣
       ↪edge type
          """
          edges_lookup = {
              'tree': set(),
              'forward': set(),
              'cross': set(),
              'back': set()
          }
          # BEGIN SOLUTION
          # Generate prev array. This time, we do need to loop through all nodes
          visited = [False]*len(adj_list)
          prev = [-1]*len(adj_list)
          # explore subroutine from earlier
          def explore(adj_list, curr):
              nonlocal visited, prev # share the same visited and prev arrays across␣
       ↪all calls to explore()

              visited[curr] = True
              for v in adj_list[curr]:
                  if not visited[v]:
                      prev[v] = curr
                      explore(adj_list, v)
          for v in range(len(adj_list)):
              if not visited[v]:
                  explore(adj_list, v)

          preorder, postorder = get_pre_post(adj_list)

          pre, post = {}, {}
          for u, time in preorder:
              pre[u] = time
          for u, time in postorder:
              post[u] = time

          for u in range(len(adj_list)):
```

```
        for v in adj_list[u]:
            edge = (u, v)
            if pre[u] < pre[v] < post[v] < post[u]:
                if prev[v] == u:
                    edges_lookup['tree'].add(edge)
                else:
                    edges_lookup['forward'].add(edge)
            elif pre[v] < pre[u] < post[u] < post[v]:
                edges_lookup['back'].add(edge)
            else:
                edges_lookup['cross'].add(edge)
    # END SOLUTION
    return edges_lookup
```

```
[ ]: grader.check("q1.3")
```

0.5 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit.

```
[ ]: grader.export(pdf=False, force_save=True, run_tests=True)
```