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=3.8` * 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**

In [1]:
```
# Install dependencies
!pip install -r requirements.txt --quiet
```

```
ERROR: pip's dependency resolver does not currently take into account all the packages that are installe
```

In [17]:
```
import otter
assert (
    otter.__version__ >= "4.4.1"
), "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("dfs-scc.ipynb")

rng_seed = 42
```

In [18]:
```
# 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**.

In [19]:
```
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
        edge_list:List[Tuple(int,int)] = edge list where each tuple (u,v) represents the direc
            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)
```

0.1.3 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.

```
In [20]: 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 ex
                 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
                 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 ex

                 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

        # 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
```

0.1.4 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.**

```
In [21]: edge_lists = []
         edge_lists.append([(0,1), (0,2), (1,2), (2,3), (3,4), (3,5), (4,5)])   # edge list of first gr
         edge_lists.append([(0,1), (0,2), (1,2), (3,4), (3,5), (4,5)])          # edge list of second g
         # 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.

In [22]:
```
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.

Points: 2

In [23]:
```
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 end
        # at the right vertices
        assert nx.is_simple_path(nx_graph, path), f"your dfs_path did not return a valid simpl
        assert path[0] == s, f"your dfs_path returned a valid simple path, but it does not sta
        assert path[-1] == t, f"your dfs_path returned a valid simple path, but it does not en

print("Success")
```

```
Testing graph: 1
Testing graph: 2
Success
```

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

0.2 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 \times$(number of vertices)$. 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.

Points: 2

```
In [10]: 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 value
            respectively. Each tuple should have a vertex as its first entry, and the pre/post
            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
```

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

0.2.1 Q2) Strongly Connected Components (Kosaraju-Sharir Algorithm)

We will now see how we can tie the concepts of DFS traversals and pre/post order values to obtain the strongly conencted components within a graph. SCC meta graphs are useful in that they allow us to construct DAGs, linearize a graph that contains cycles and obtain equivalence classes within a graph among other use cases.

S. Rao Kosaraju and Micha Sharir independently solved this problem of finding the SCCs within a graph with the Kosaraju-Sharir Algorithm (which is the algorithm presented in the DPV notes section 3.4.2).

A high-level overview of the algorithm is as follows: 1. Run DFS on the reversed graph and store the post numbers of each node. 2. Run DFS on the original graph, visiting nodes in decreasing post number order. Each DFS tree in this traversal is an SCC.

The first thing we need is a reversed graph G^R. In this part, we'll implement a function to reverse a graph represented as an adjacency list.

> **Task:** Given a graph represented as an adjacency list, return the acjacency list that represents the reversed graph (the vertex set is the same as the input graph, the edge set contains (v, u) if and only if (u, v) is present in the input graph's edge set).

Points: 1

```
In [12]: def reverse_graph(adj_list):
             """
             args:
                 adj_list:List[List[int]] = the adjacency list that represents our input graph
             return:
                 List[List[int]] representing the adjacency list of the reversed input graph
             """
             # BEGIN SOLUTION
             reversed_adj_list = [[] for _ in range(len(adj_list))]
             for u in range(len(adj_list)):
                 for v in adj_list[u]:
                     reversed_adj_list[v].append(u)
             return reversed_adj_list
             # END SOLUTION
```

```
In [ ]: grader.check("q2.1")
```

Now, you get to complete the rest of the algorithm to find the SCCs within a graph.

> **Task:** * Given a graph represented as an adjacency list, return a list of SCC components. Return SCCs in the order that they are found. * The SCC components should be represented as Python `Set`s that contain only the nodes present in that SCC.

Points: 2

```
In [14]: def find_SCCs(adj_list):
             """
             args:
                 adj_list:List[List[int]] = the adjacency list that represents our input graph
             return:
                 List(Set(int, ...), ...) a list of sets where each set contains all the nodes
                     that belong to the corresponding SCC
             """
             scc_list = []
```

```
    # YOUR CODE HERE
    # BEGIN SOLUTION
    reversed_graph = reverse_graph(adj_list)
    _, postorder = get_pre_post(reversed_graph)
    visited = [False]*len(adj_list)

    def explore(u):
        visited[u] = True
        curr_set.append(u)
        for v in adj_list[u]:
            if not visited[v]:
                explore(v)

    for u, _ in postorder[::-1]:
        if not visited[u]:
            curr_set = []
            explore(u)
            scc_list.append(set(curr_set))
    # END SOLUTION

    return scc_list
```

```
In [ ]: grader.check("q2.2")
```

0.3 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.

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