CS 170 Homework 4

Due **9/27/2023, 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, you must explicitly write "none".

2 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 $s[v]$ be the number of descendants of v that are a distance of at least K from v. Describe an $O(|V|)$ algorithm to output $s[v]$ for every v. Give a 3-part solution.

Solution:
Observe that for a vertex v,

$$s[v] = \sum_{c \text{ is a child of } v} s[c] + (\#\text{ descendants of } v \text{ at distance } K) \tag{1}$$

The algorithm proceeds as follows. Initialize all the $s[v]$ 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 $s[a]$ to account for v. Once we finish processing a child c of v, we increment $s[v]$ by $s[c]$.

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

Correctness For a vertex v and a child c, every node counted in $s[c]$ should be included in $s[v]$ 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 $s[v]$. Notice that when we finish processing a node v, its $s[v]$ value will be correct and so it can be used by its parent.

3 Where's the Graph?

Each of the following problems can be solved with techniques taught in lecture. Construct a simple directed graph, write an algorithm for each problem by black-boxing algorithms taught in lecture, and analyze its runtime. You do not need to provide proofs of correctness.

(a) The CS 170 course staff is helping build a roadway system for PNPenguin's hometown in Antarctica. Since we have skill issues, we are only able to build the system using one-way roads between igloo homes. Before we begin construction, PNPenguin wants to evaluate the reliability of our design. To do this, they want to determine the number of *reliable* igloos; an igloo is reliable if you are able to leave the igloo along some road and then return to it along a path of other roads. However, PNPenguin isn't very good at algorithms, and they need your help.

Given our proposed roadway layout, design an efficient algorithm that determines the number of reliable igloos.

Solution:

Main Idea: Construct a directed graph $G = (V, E)$ with vertices representing the igloos and edges representing the one-way roads. Then, a reliable igloo is equivalent to a vertex that belongs to a cycle. Thus, our job is to determine the number of vertices in G that belong to some cycle.

First, we run Kosaraju's algorithm on G to compute all of its strongly connected components and slightly modify the algorithm so that it also computes the number of vertices in each SCC. Then, the desired set of vertices can be obtained by including all vertices $v \in V$ that lie in an SCC with size at least 2.

Proof of Correctness: Any SCC $S \subset G$ is a subgraph in which all its vertices $v \in S$ can reach one another. Thus, the vertices of any SCC with size at least two by definition have a cycle because any vertex v can reach any other vertex $u \in v \in S$, which in turn can return to v.

Runtime Analysis: Running Kosaraju's takes $O(n+m)$ time, and traversing through all of the vertices to determine the desired set of vertices takes $O(n)$ time. This is assuming that during Kosaraju's, we construct a hashmap with vertices as keys and SCC size as values. Thus, the overall runtime is

$$O(n+m) + O(n) \in O(n+m)$$

(b) There are p different species of Pokemon, all descended from the original Mew. For any species of Pokemon, Professor Juniper knows all of the species *directly* descended from it. She wants to write a program that answers queries about these Pokemon. The program would have two inputs: a and b, which represent two different species of Pokemon. Her program would then output one of three options in constant time (the time complexity cannot rely on p):

(1) a is descended from b.

(2) b is descended from a.

(3) a and b share a common ancestor, but neither are descended from each other.

Unfortunately, Professor Juniper's laptop is very old and its SSD drive only has enough space to store up to $O(p)$ pieces of data for the program. Give an algorithm that Professor Juniper's program could use to solve the problem above given the constraints.

Hint: Professor Juniper can run some algorithm on her data before all of her queries and store the outputs of the algorithm for her program; time taken for this precomputation is not considered in the query run time.

Solution: This is a directed graph, with each node representing some species and an edge from x to y indicating that y descended *directly* from x. You can think of the graph as a directed tree rooted at Mew.

We run DFS on this graph, storing the post-numbers and pre-numbers for each node. When the program is queried, it checks whether a is descended from b by checking if $pre[b] < pre[a] < post[a]$, or b is descended from a ($pre[a] < pre[b] < post[a]$), otherwise concludes a and b share a common ancestor but are not descended from each other.

The DFS takes $O(p)$ time, but it is part of the precomputation. The query involves indexing into two arrays and comparing them, all of which takes $O(1)$ time.

(c) Bob has n different boxes. He wants to send the famous "Blue Roses' Unicorn" figurine from his glass menagerie to his crush. To protect it, he will put it in a sequence of boxes. Each box has a weight w and size s; with advances in technology, some boxes have negative weight. A box a inside a box b cannot be more than 15% smaller than the size of box b; otherwise, the box will move, and the precious figurine will shatter. The figurine needs to be placed in the smallest box x of Bob's box collection.

Bob (and Bob's computer) can ask his digital home assistant Falexa to give him a list of all boxes less than 15% smaller than a given box c (i.e. all boxes that have size between 0.85 to 1 times that of c). Bob will need to pay postage for each unit of weight (for negative weights, the post office will pay Bob!). Find an algorithm that will find the lightest sequence of boxes that can fit in each other in linear time (in terms of the graph).

Hint: How can we create a graph knowing that no larger box can fit into a smaller box, and what property does this graph have?

Solution: We can view each box as a node in a directed graph, with an edge from a to b indicating that a fits in b. Since Falexa only gives the edges in the opposite direction (in that it tells Bob the boxes that can fit into b), we have to construct a graph and then reverse the edges, which is doable in linear time. This graph is a DAG. If box a can fit in box b and box c can fit in box a, box b cannot fit into box c. We set incoming edge weights to a node a to be the weight w of the corresponding box. We can linearize this DAG by running DFS starting at the smallest box x, and then find the shortest path from the x to any other box in linear time by iterating through the DAG's vertices in topological order to compute each new shortest path by building on paths to previous nodes in the DAG.

4 Semiconnected DAG

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.

5 2-SAT

In the 2SAT problem, you are given a set of clauses, where each clause is the disjunction (OR) of two literals (a literal is a Boolean variable or the negation of a Boolean variable). You are looking for a way to assign a value true or false to each of the variables so that all clauses are satisfied – that is, there is at least one true literal in each clause. For example, here's an instance of 2SAT:

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee x_4)$$

Recall that $\vee$ is the logical-OR operator and $\wedge$ is the logical-AND operator and $\overline{x}$ denotes the negation of the variable x. This instance has a satisfying assignment: set x_1, x_2, x_3, and x_4 to `true, false, false, and true`, respectively.

The purpose of this problem is to lead you to a way of solving 2SAT efficiently by reducing it to the problem of finding the strongly connected components of a directed graph. Given an instance I of 2SAT with n variables and m clauses, construct a directed graph $G_I = (V, E)$ as follows.

- G_I has $2n$ nodes: one for each variable and its negation.
- G_I has $2m$ edges: for each clause $(\alpha \vee \beta)$ of I (where α, β are literals), G_I has an edge from from $\overline{\alpha}$ to β, and one from the $\overline{\beta}$ to α.

Note that the clause $(\alpha \vee \beta)$ is equivalent to each of the implications $\overline{\alpha} \implies \beta$ and $\overline{\beta} \implies \alpha$. In this sense, G_I records all implications in I.

(a) Show that if G_I has a strongly connected component containing both x and $\overline{x}$ for some variable x, then I has no satisfying assignment.

(b) Now show the converse of (a): namely, that if none of G_I's strongly connected components contain both a literal and its negation, then the instance I must be satisfiable.

Hint: Pick a sink SCC of G_I. Assign variable values so that all literals in the sink are True. Why are we allowed to do this, and why doesn't it break any implications?

(c) Conclude that there is a linear-time algorithm for solving 2SAT. Provide the algorithm description and runtime analysis; proof of correctness is not required.

Solution:

(a) Suppose there is a SCC containing both x and $\overline{x}$. Notice that the edges of the graph are necessary implications. Thus, if some x and $\overline{x}$ are in the same component, there is a chain of implications which is equivalent to $x \to \overline{x}$ and a different chain which is equivalent to $\overline{x} \to x$, i.e. there is a contradiction in the set of clauses.

(b) Take any sink component, and assign variables so all the literals in this component are True. Because of how we define the graph, there is a corresponding source component which has the negations of all literals in this component. Remove this source/sink component pair, and repeat the process until the graph is empty. Since we set components to true in reverse topological order, there is no implication from a true literal to a false literal. Since no literal and its negation are in the same SCC, we never try to set a variable to be both true and false. So this produces an assignment satisfying all clauses.

(c) Let φ be a formula acting on n literals $x_1, \ldots, x_n$. Construct a graph with $2n$ vertices representing the set of literals and their negations. For each clause $(a \vee b)$ of φ add the edges $\overline{a} \Rightarrow b$ and $\overline{b} \Rightarrow a$. Use the strongly connected components algorithm and for each i, check if there is a SCC containing both x_i and $\overline{x_i}$. If any such component is found, report unsatisfiable. Otherwise, report satisfiable.

(Note: A common mistake is to report unsatisfiable if there is a path from x_i to $\overline{x_i}$ in this graph, even if there is no path from $\overline{x_i}$ to x_i. Even if there is a series of implications which combined give $x_i \to \overline{x_i}$, unless we also know $\overline{x_i} \to x_i$ we could set x_i to False and still possibly satisfy the clauses. For example, consider the 2-SAT formula $(\overline{a} \vee b) \wedge (\overline{a} \vee \overline{b})$. These clauses are equivalent to $a \to b, b \to \overline{a}$, which implies $a \to \overline{a}$, but this 2-SAT formula is still easily satisfiable.)

6 [Coding] DFS and Kosaraju's Algorithm

For this week's coding questions, we'll implement DFS and apply it to find the strongly connected components within a graph via Kosaraju's algorithm. There are two ways that you can access the notebook and complete the problems:

1. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   `https://github.com/Berkeley-CS170/cs170-fa23-coding`

   and navigate to the `hw04` folder. Refer to the `README.md` for local setup instructions.

2. **On Datahub**: Click here and navigate to the `hw04` folder if you prefer to complete this question on Berkeley DataHub.

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the gradescope assignment titled "Homework 4 Coding Portion".
- *OH/HWP Instructions:* Designated coding course staff will provide conceptual and debugging help during office hours and homework parties.
- *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.

dfs_scc

September 22, 2023

0.1 Depth First Search and Strongly Connected Components

[3]:
```
# Install dependencies
!pip install -r requirements.txt --quiet
```

[1]:
```
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
from tqdm import tqdm
import pickle
grader = otter.Notebook("dfs_scc.ipynb")

rng_seed = 42
```

[2]:
```
# 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**.

[3]:
```python
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
```

0.1.1 Q1) Depth First Search

In this question, you will have to implement a DFS traversal for the given adj_list. As you traverse the graph, you should update a prev array, where prev[v] = u if you came to v from u. Return this list prev. By default prev[start] should equal start.

For all subparts, your DFS should prioritize visiting lower valued vertices first.

> **Task 1.1** * Given a graph represented as an adjacency list, perform a DFS traversal that prioritizes visiting the lower valued nodes first * Return a list of length=number of nodes where the ith element in this list contains the node that explored node i. If node i is the root of the dfs tree, then the ith element should be node i. * Make sure to traverse every node present in the graph, even if the graph is disconnected.

Points: 2

[12]:
```python
def dfs(adj_list):
    """
    args:
        adj_list:List[List[int]] = the adjacency list that represents our input␣
↪graph
    return:
        A List[int] where the ith element represents the node that called␣
↪dfs(i). If a
            DFS is started from a vertex, then List[i] = i.
    """
    # BEGIN SOLUTION
    visited = set()
    prev = [-1] * len(adj_list)
    def explore(u):
        visited.add(u)
```

```
        for v in adj_list[u]:
            if v not in visited:
                prev[v] = u
                explore(v)
    for start in range(len(adj_list)):
        if start not in visited:
            prev[start] = start
            explore(start)
    return prev
    # END SOLUTION
```

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

Pre and Postorder Values In class we showed how to use DFS to check if there exists a path between two nodes, topologically sort nodes, and find SCC's. In those algorithms, pre and post numbers were crucial.

> **Task 1.2** * Rework your implementation of DFS in this next cell to allow it to generate pre and post order numbers for each node. * Your smallest preorder number should be 1. Your largest postorder number should be $ 2 \times$(number of vertices)$. * Return two lists of tuples. Each list should contain a tuple of two values Tuple(Node, Time Visited). The first list should contain the tuples with the preorder visits, and the second list should contain the tuples with the postorder visits. * We have provided some boiler plate code to get you started. Feel free to write your own if you find that easier. * Feel free to refer to Figure 3.6(b) in section 3.2.3 of the DPV notes for how your pre and post order values should look like. >* If the input was the left most component, you should return [(A, 1), (B, 2), (E, 4), (I, 5), (J, 6)], [(B, 3), (J, 7), (I, 8), (E, 9), (A, 10)]. >* Both lists should be sorted according to the second element in the tuple.

Points: 2

[25]:
```
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 = []
    # BEGIN SOLUTION
    visited = set()
```

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

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

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.

Points: 2

[47]:
```
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
    prev = dfs(adj_list)
    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")
```

0.1.2 Q2) Strongly Connected Components

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 solved this problem of finding the SCCs within a graph in 1978 with the Kosaraju-Sharir Algorithm (which is the algorithm presented in the DPV notes section 3.4.2). The first step of this algorithm is to reverse the graph.

> **Task 2.1** * 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: 2

[51]:
```
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
    """
    reversed_adj_list = [[] for i in range(len(adj_list))]
    # BEGIN SOLUTION
    for u in range(len(adj_list)):
        for v in adj_list[u]:
            reversed_adj_list[v].append(u)
    # END SOLUTION
    return reversed_adj_list
```

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

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

> **Task 2.2** * Given a graph represented as an adjacency list, return a list of SCC components. * The SCC components should be represented as sets that contain only the nodes present in that SCC.

Points: 2

[34]:
```
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 = []
    # BEGIN SOLUTION
    reversed_graph = reverse_graph(adj_list)
    _, postorder = get_pre_post(reversed_graph)
    visited = set()

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

    for u, _ in postorder[::-1]:
```

```
        if u not in visited:
            curr_set = []
            explore(u)
            scc_list.append(set(curr_set))
    # END SOLUTION
    return scc_list
```

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

0.2 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)
```