CS 170 Homework 3

Due **9/20/2021, at 10:00 pm**

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.

In addition, we would like to share correct student solutions that are well-written with the class after each homework. Are you okay with your correct solutions being used for this purpose? Answer "Yes", "Yes but anonymously", or "No"

2 Updating Labels

You are given a tree $T = (V, E)$ with a designated root node r, and a non-negative integer label $l(v)$ for each node v. We wish to relabel each vertex v, such that $l_{\text{new}}(v)$ is equal to $l(w)$, where w is the kth ancestor of v in the tree for $k = l(v)$. We follow the convention that the root node, r, is its own parent. Give a linear time algorithm to compute the new label, $l_{\text{new}}(v)$ for each v in V

Slightly more formally, the *parent* of any $v \neq r$, is defined to be the node adjacent to v in the path from r to v. By convention, $p(r) = r$. For $k > 1$, define $p^k(v) = p^{k-1}(p(v))$ and $p^1(v) = p(v)$ (so p^k is the kth ancestor of v). Each vertex v of the tree has an associated non-negative integer label $l(v)$. We want to find a linear-time algorithm to update the labels of all vertices in T according to the following rule: $l_{\text{new}}(v) = l(p^{l(v)}(v))$.

Describe the algorithm and give a runtime analysis; no proof of correctness is necessary.

Solution:

Main Idea At every step of the algorithm, we will maintain the labels of 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 once. Since a path can have at most n vertices, the length of this array is at most n. Once we've processed all the children of a node, we can index into the array and set its label equal to the index of its kth ancestor. Notice that if we relabel the vertex before processing its children, we overwrite a label that the children of the vertex could depend on.

Runtime Analysis Since we add only a constant number of operations at each step of DFS, the algorithm is still linear time.

3 Disrupting a Network of Spies

Let $G = (V, E)$ denote the "social network" of a group of spies. In other words, G is an undirected graph where each vertex $v \in V$ corresponds to a spy, and we introduce the edge $\{u, v\}$ if spies u and v have had contact with each other. The police would like to determine

which spy they should try to capture, to disrupt the coordination of the group of spies as much as possible. More precisely, the goal is to find a single vertex $v \in V$ whose removal from the graph splits the graph into as many different connected components as possible. This problem will walk you through the design of a linear-time algorithm to solve this problem. In other words, the running time will be $O(|V| + |E|)$.

In the following, let $f(v)$ denote the number of connected components in the graph obtained after deleting vertex v from G. Also, assume that the initial graph G is connected (before any vertex is deleted), has at least two vertices, and is represented in an adjacency list format.

For each part, prove that your answer is correct (some parts are simple enough that the proof can be a brief justification; others will be more involved).

(a) Let T be a tree produced by running DFS on G with root $r \in V$. (In particular, $T = (V, E_T)$ is a spanning tree of G: a tree which connects all the nodes of G using a subset of its edges.) Given T, find an efficient way to calculate $f(r)$.

(b) Let $v \in V$ be some vertex that is not the root of T (i.e., $v \neq r$). Suppose further that there is no edge $\{u, w\}$ such that u is a descendant of v and w is an ancestor of v. How could you calculate $f(v)$ from T in an efficient way?

(c) For $w \in V$, let $D_T(w)$ be the set of descendants of w in T including w itself. For a set $S \subseteq V$, let $N_G(S)$ be the set of *neighbors* of S in G, i.e. $N_G(S) = \{y \in V : \exists x \in S \text{ s.t. } \{x, y\} \in E\}$. We define $\mathsf{up}_T(w) := \min_{y \in N_G(D_T(w))} \mathsf{depth}_T(y)$, i.e. the smallest depth in T of any neighbor in G of any descendant of w in T.

Now suppose v is an arbitrary non-root node in T, with children $w_1, \ldots, w_k$. Describe how to compute $f(v)$ as a function of k, $\mathsf{up}_T(w_1), \ldots, \mathsf{up}_T(w_k)$, and $\mathsf{depth}_T(v)$.

Hint: Think about what happened in part (b); think about what changes when we can have non-tree edges that go up from one of v's descendants to one of v's ancestors, and think about how you can detect it from the information provided.

(d) Design an algorithm which, on input G, T, computes $\mathsf{up}_T(v)$ for all vertices $v \in V$, in linear time.

(e) Given G, describe how to compute $f(v)$ for all vertices $v \in V$, in linear time.

Solution:

Lemma: Let T be a DFS tree, and let $u, v \in V$ be such that u is neither a descendant nor an ancestor of v in T. Then there is no edge $\{u, v\} \in E$.

Proof: Suppose that there is an edge $\{u, v\} \in E$, and suppose that u is visited first in the DFS. Then at some point we leave u without traversing $\{u, v\}$ (else v would be a descendant of u). But this means that v was visited between entering and leaving u, and so it is a descendant of u. If v was visited first, the same argument shows that v would be an ancestor of u.

(a) $f(r) =$ the number of children of r.

Proof: Let k be the number of children of r, and let $T_1, \ldots, T_k$ be the subtrees of T rooted at those children. If $u \in T_i, v \in T_j$ for $i \neq j$, then the conditions of the lemma hold and so $\{u, v\} \notin E$, and so each T_i is a connected component when we remove r.

(b) $f(v) = 1+$ the number of children of v.

Proof: Consider a partition of V into three sets $\{A, B, C\}$, where A is the set of ancestors of v, B is the tree rooted at v, and C is the rest of the graph. It suffices to show that there are no edges from $B - v$ to $A \cup C$; since $A \cup C$ is connected, applying the previous subpart to B gives the result. For every $u \in B - v$, all descendants and ancestors of u lie in $A \cup B$, and so by the lemma there is no edge from u to C. By assumption there are no edges from u to A, and so $B - v$ has no edges to $A \cup C$.

(c) Let N denote the number of children c of v with the property that $\mathsf{up}_T(c) \geq \mathsf{depth}(v)$, i.e., $N = |\{w : w \text{ is a child of } v \text{ and } \mathsf{up}_T(c) \geq \mathsf{depth}(v)\}|$. Then $f(v) = N + 1$.

Proof: If we show that a child c has $\mathsf{up}_T(c) < \mathsf{depth}(v)$ iff c is connected to an ancestor of v by a path that excludes v, then the proof follows directly from (b) because these children are in the same connected component as the root r.

If c is connected to a proper ancestor a of v by a path that excludes v then $\mathsf{up}_T(c) \leq \mathsf{depth}(a) < \mathsf{depth}(v)$. Conversely, if $\mathsf{up}_T(c) < \mathsf{depth}(v)$, then there is an edge from a descendant d of v to some vertex w which has smaller depth than v. Since w cannot be a descendant of d, it must be an ancestor of d by the lemma, and since it has smaller depth than v, it is also an ancestor of v.

(d) By definition, $\mathsf{up}_T(v)$ is the minimum of v's neighbors' depths, and the up_Ts of v's descendants. Formally,

$$\mathsf{up}_T(v) = \min\big(\min\{\mathsf{depth}(w) : \{v, w\} \in E\},\ \min\{\mathsf{up}_T(w) : w \text{ is a child of } v\}\big).$$

We can thus compute $\mathsf{up}_T(v)$ by traversing the DFS tree bottom-up (e.g. in depth-first order, computing $\mathsf{up}_T(v)$ at the end of $\mathsf{visit}(v)$).

For a leaf, $\mathsf{up}_T(v)$ can be computed by minimizing over the depth of all neighboring vertices.

This can be computed in linear-time as each vertex and edge is considered a constant number of times.

(e) This follows immediately from parts (c)–(e). Pick some node as the root. Then compute $\mathsf{up}_T(\cdot)$ and $\mathsf{depth}(\cdot)$ at each node. Then, compute the function defined in part (d).

The running time is $\Theta(|V| + |E|)$. We need to make three passes over the graph: one to compute $\mathsf{depth}(\cdot)$, one to compute $\mathsf{up}_T(\cdot)$, and a third to compute $f(\cdot)$. In each pass, we process each vertex once, and each edge at each vertex. This is $\Theta(|V| + |E|)$ for each pass, and $\Theta(3|V| + 3|E|) = \Theta(|V| + |E|)$.

Note that in a practical implementation, some of these separate passes could be combined, without affecting the asymptotic complexity of the algorithm.

4 Where's the graph?

Each of the following problems can be solved with techniques taught in lecture. Construct a simple directed graph and write an algorithm for each problem by black-boxing algorithms taught in lecture and in the textbook.

(a) Sarah wants to do an extra credit problem for her math class. She is given three numbers: 1, x, and y. Starting from x, she needs to find the shortest sequence of additions, subtractions, and divisions (only possible when the number is divisble by y) using 1 and y to get to 2021. If there are multiple sequences with the shortest length, return any one of them. She can use 1 and y multiple times. Give an algorithm that Sarah can query to get this sequence of arithmetic operations.

Solution: We can view this as a BFS problem, where the nodes in the graph are numbers that have been calculated. There are at most five edges from each node: $+1$, -1, $+y$, $-y$, and $/y$. Our algorithm starts from x and run BFS on this graph until the node 2021 is reached.

(b) There are n different species of Gem Berry, all descended from the original Stone Berry. For any species of Gem Berry, Emily knows all of the species directly descended from it. Emily wants to write a program. There would be two inputs to her program: a and b, which represent two different species of Gem Berries. Her program will then output one of three options in constant time (the time complexity cannot rely on n):

(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.

Give an algorithm that Emily's program could use to do this.

Solution: This is a directed graph, with each node representing some species and an edge from x to y indicating that y descended from x. We run DFS on this graph, storing the post-numbers and pre-numbers for each node. When the program is queried, it checks whether the edge (a, b) is a back edge (a is descended from b), a tree edge (b is descended from a), or a cross edge (a and b share a common ancestor but are not descended from each other)

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:

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

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 the negation of α to β, and one from the negation of β to α.

Note that the clause $(\alpha \vee \beta)$ is equivalent to either of the implications $\overline{\alpha} \implies \beta$ or $\overline{\beta} \implies \alpha$. 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*: Assign values to the variables as follows: repeatedly pick a sink strongly connected component of G_I . Assign value `true` to all literals in the sink, assign `false` to their negations, and delete all of these. Show that this ends up discovering a satisfying assignment.)

(c) Conclude that there is a linear-time algorithm for solving 2SAT.

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 Introduction To Coding Homework

In this iteration of CS170, we will be introducing coding problems that will be conducted on an Online Judge platform in the style of programming contests (e.g Leetcode, Codeforces). This will only be a small part of the homework (roughly 1 problem on only 3 of the homeworks, so 3 total in the semester).

We will be using QDUOJ, an open-source online judge platform, to test your solutions. To access this platform, visit hellfire.ocf.berkeley.edu).

Rules

(a) Any rules described on the Syllabus are in effect.

(b) You may refer to online resources for minor implementation details, such as searching up documentation for a heap class. You may not consult any resources that give away the main idea or implementation of the algorithm.

(c) Similar to the written homework, you may not collaborate on the implementation details and must submit your own code.

(d) You may not exploit bugs on the platform, access otherwise not publicly visible test cases, or submit malicious code.

Some Details

(a) Languages
For fairness, we will only allow Python 3.

(b) Input & Output
We will use stdin and stdout for input and output. The format will be given and you do not need to consider invalid inputs.

(c) If you receive a "Time Limit Exceeded!" message, it is most likely an issue with the efficiency of your implementation, as all time limits are tested on the same hardware.

Action items for this homework

(a) Log in to the contest system and register **with your bMail address as it appears on Gradescope**. You can use any nickname you wish as long as the email address matches your bMail address on Gradescope. Note that this username may be displayed to other students if you wish to opt-in to the leaderboard.

(b) Make a test submission to the problem below to familiarize yourself with the platform.

A+B Problem

Access the problem at: https://hellfire.ocf.berkeley.edu/contest. Choose the appropriate contest (Homework 3 Coding Problems, or the DSP 150% or DSP 200% alternative), navigate to "Problems" on the toolbar on the right and choose the **standard** version of the A+B problem. Feel free to ignore the Contest version of the problem.
Your task is to make any submission. You will not be penalized if your solution is wrong, but you have to submit some code for this problem. In the case if you encounter a bug with the online contest system, for example, you are having problems with registration or problems with submitting, please, report the bug to this form: https://forms.gle/17aykrtBas5CmiV26.

Description

To practice basic input/output and to familiarize with this OJ platform.
Input n, then on the following n lines two nonnegative integers A and B, output the sum of these two nonnegative integers $A + B$.

Input

The first line will be one positive integer The following Of 10% of the test cases $0 < A, B \leq 10$, and it is guaranteed that $n > 0 > 200$.
Of 100% of the test cases $0 < A, B \leq 200$, and it is guaranteed that $n > 0 > 200$.

Output

On each line, print out the sum of the two integers given.

Sample Input

3
114 51
4 19
198 10

Sample Output

165
23
208

Submission

Make a screenshot of your submission and attach it to your homework PDF, where you can submit it on gradescope with the rest of the problems. In the future, we intend on building a gradescope integration, so you may not need to do this for future homeworks.